Demaq System Documentation and User Manual

Alexander Böhm

March 8, 2010

Contents

1 Introduction **4**

2 Fundamental Concepts **5**
- 2.1 Traditional Approach . . . 5
- 2.2 Demaq Approach . . . 6
- 2.3 Differences . . . 7
  - 2.3.1 Handling XML data . . . 7
  - 2.3.2 Persistent Data Storage . . . 7
  - 2.3.3 Application Code . . . 8
  - 2.3.4 Runtime Context and Instances . . . 8

3 Queue Definition Language **10**
- 3.1 Queues . . . 10
  - 3.1.1 Gateway queues . . . 10
  - 3.1.2 Basic queues . . . 12
  - 3.1.3 Advanced features . . . 12
- 3.2 Properties . . . 14
  - 3.2.1 Computed properties . . . 15
  - 3.2.2 Fixed properties . . . 15
  - 3.2.3 Inherited properties . . . 16
- 3.3 Slicings . . . 17
  - 3.3.1 Using a slicing to partition a single queue . . . 17
  - 3.3.2 Using a slicing on several input queues . . . 18
  - 3.3.3 Using slicings in application programs . . . 19
  - 3.3.4 Slice content visibility . . . 20

4 Queue Rule Language **23**
- 4.1 Overview . . . 23
  - 4.1.1 A first example: Hello, world . . . 23
  - 4.1.2 Writing rule bodies with XQuery . . . 24
  - 4.1.3 Rule definition . . . 25
- 4.2 Data Access Functions . . . 25
  - 4.2.1 Retrieving the context item . . . 26
  - 4.2.2 Retrieving all messages in a queue . . . 26

4.2.3 Retrieving the slicekey . . . 27
4.2.4 Retrieving the messages in a slice . . . 28
4.2.5 Retrieving the value of a property . . . 29
4.2.6 Retrieving the timestamp of a message . . . 30
4.2.7 Retrieving the unique ID (messageID) of a message . . . 30
4.2.8 Creating a unique identifier within a rule . . . 31
4.3 Enqueue message expression . . . 32
4.3.1 Enqueuing XML fragments into a queue . . . 32
4.3.2 Defining message properties . . . 33
4.3.3 Performing delayed message enqueuing . . . 33
4.3.4 Sending messages over a gateway queue . . . 34
4.3.5 Handling incoming HTTP GET requests . . . 35
4.3.6 System-provided properties for gateway queues . . . 36
4.3.7 Correlating reply messages to incoming requests . . . 37
4.4 Additional Demaq Updating Expressions . . . 38
4.4.1 Triggering System Shutdown . . . 38
4.4.2 Tracking System Activity . . . 38
4.4.3 Requesting Garbage Collection . . . 39
4.5 Error Handling . . . 39
4.5.1 Default error queue . . . 40
4.5.2 Queue-specific error handlers . . . 40
4.5.3 Rule-specific error handlers . . . 40
4.5.4 Error queue selection . . . 41
4.5.5 Error Message Format . . . 42
4.6 Processing model . . . 43
4.7 Application Modularization . . . 44
4.7.1 Module Design . . . 44
4.7.2 Application Module Specification . . . 45
4.7.3 Module Import and Instantiation . . . 46
4.8 Debugging applications . . . 47
4.8.1 Calling trace methods in application rules . . . 47
4.8.2 Detecting runtime errors . . . 48

5 Application Deployment and Runtime 50
5.1 Deployment Steps . . . 50
5.1.1 Instance Creation . . . 50
5.1.2 Importing the Application . . . 53
5.1.3 Instance Startup . . . 53
5.1.4 Instance Shutdown . . . 53
5.1.5 Closing an Instance . . . 53
5.1.6 Destroying an Instance . . . 53
5.2 Application Runtime . . . 53
5.2.1 Garbage Collector . . . 54
5.2.2 System Trace . . . 54
5.2.3 Interactive Debugger . . . 55

6 System Installation **56**
6.1 Required Third-Party Software Packages . . . 56
6.2 Retrieving the Source Code . . . 57
6.3 Configuration and Compilation . . . 57
6.3.1 Setting up a build directory . . . 57
6.3.2 Configuring the build directory . . . 57
6.3.3 Performing an initial build . . . 58
6.3.4 Setting environment variables . . . 58
6.4 Compile-time Configuration Options . . . 59
6.4.1 Building Demaq with IBM DB/2 as Message Store . . . 60
6.5 Speeding up the build process . . . 61

7 System Architecture **62**
7.1 Query Compiler . . . 62
7.2 Runtime System . . . 62
7.2.1 XML Message Storage . . . 63
7.2.2 Runtime Core . . . 63
7.2.3 Communication System . . . 64
7.3 Visual Editor . . . 64

8 Implementation **66**
8.1 Query Compiler . . . 66
8.1.1 Query Rewrites . . . 66
8.1.2 AST Serialization / Execution Plan . . . 70
8.2 Runtime System . . . 71
8.2.1 XML Message Storage . . . 71
8.2.2 Runtime Core . . . 71
8.2.3 Communication System . . . 72
8.3 Test Framework . . . 72
8.3.1 Running Tests . . . 72
8.3.2 C++ Unit Tests . . . 73
8.3.3 Application Tests . . . 73

9 Legion Application Distribution System **75**
9.1 Dependency Analysis . . . 75
9.2 Host Allocation . . . 75
9.3 Code Generation . . . 76
9.4 Scalability Transformations . . . 76

10 Further information **77**

A Frequently Asked Questions **78**
A.1 General . . . 78
A.2 Application Developers . . . 78
A.3 Demaq Hackers . . . 79

Chapter 1

Introduction

Welcome to Demaq, a system for **DE**clarative **M**essaging **A**nd **Q**ueuing. The goal of the Demaq system is to create a programming and execution environment for distributed applications that are based on asynchronous XML message exchange. Examples for this kind of applications include Web Services based on SOAP or REST, AJAX applications, RSS feeds, etc.

Despite the application servers and imperative programming languages (e.g. Java or C++) that are common today, Demaq describes the logic of an application instance using a declarative language. Examples for such declarative languages are SQL or XQuery, which is used as the basis for the Demaq programming language. Declarativity basically means that instead of requiring developers to provide a detailed execution plan telling the runtime system the individual steps that should be performed, developers only have to provide a very high-level specification of the application logic. Thus - hopefully - application development becomes easier. Additionally, the declarative specification also allows the execution system to automatically apply optimizations, thus speeding up runtime performance without requiring manual tuning by the developer.

The purpose of this guide is to provide a brief overview of the Demaq system, including its declarative programming language DQL that is used to implement Demaq applications. The primary goal is to give application developers creating XML messaging applications enough information to use the Demaq system for implementing their applications. Thus, language constructs and the novel programming style are discussed extensively. Additionally, this guide aims at giving an overview of the internals of the Demaq system for hackers joining the Demaq team and potentially contributing to the project. This part of the guide is rather brief, only discussing the fundamental concepts and leaving a detailed introduction to the source code (and the doxygen-generated documentation).

Throughout this guide, we will use an online shop (e.g. Amazon, Ebay, ...) as a running example. This shop exclusively communicates with remote parties (customers, suppliers, bank, ...) using Web Services.

Chapter 2

Fundamental Concepts

The way applications are developed in Demaq significantly differs from the approaches taken by most of today's systems. In this chapter, we will review the way applications are build today (Section 2.1), before discussing the approach taken in Demaq in Section 2.2. Section 2.3 summarizes the difference and some of the benefits and drawbacks of both approaches.

2.1 Traditional Approach

Usually, our online shop example would be build using an imperative, object-oriented programming language (e.g. Java), and deployed on a corresponding application server (Tomcat, Oracle, ...). To store data persistently (customer orders, catalogs), a relational database management system would be applied. To allow accessing data as Java objects instead of writing SQL code, middleware managed persistence solutions or mapping tools (e.g. Hibernate) could be used. The required Web Service features and communication would be performed by a corresponding module of the application server (e.g. by using Apache AXIS). Figure 2.1 visualizes this architecture.

While there is no uniform design for the application implementing the business logic of the online store, most applications rely on the concept of *contexts* or *application instances*. An application server may run several of these instances in parallel (e.g. one for each active connection), each of them containing customer-specific data (e.g. the shopping cart, the master data for the customer, etc.). These data is usually represented by (context-) local variables. Once an update has to be performed (the customer send a request to add a new item to its cart, or wants to change her address master data), the context is changed using in-place updates of the context variables.

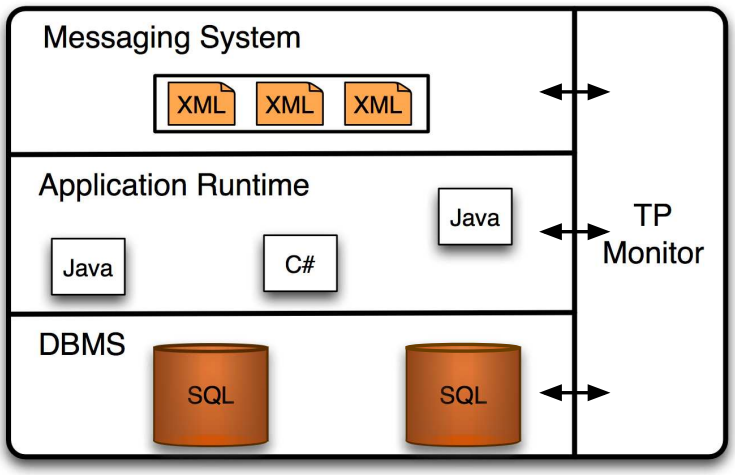

Figure 2.1: N-tier middleware architecture

2.2 Demaq Approach

The idea of the Demaq system is to implement the business logic of an XML-processing application (such as our online shop) without application contexts and auxiliary modules. Instead, the application directly operates on the XML messages that are received from external parties and performs the corresponding operations. Due to the use of Web Services, both incoming and outgoing operations are exclusively performed using XML messages. Thus, fundamentally, the idea of Demaq is to describe an XML processing application by a set of queries, transforming input XML messages into output/reply XML message.

For this purpose, Demaq uses its own, declarative and XML-aware programming language (DQL), that is build on the foundation of XQuery. Using DQL the entire business logic of a distributed application is described by a set of XQuery fragments that are evaluated on the messages received from external communication endpoints. These query fragments yield new XML messages that are sent as a reply, or to other remote systems.

Demaq uses *XML message queues* to communicate with external systems. Whenever an incoming request is received, the corresponding XML message is stored in a queue, before it is being processed. Similarly, messages that should be sent to external systems are placed into (outgoing) message queues, from where they are transferred. To express which query fragments should be used for the messages in a particular queue (e.g. representing customer orders), Demaq uses the concept of *rules*. These rules associate a query fragment to a particular queue, and also define the target queue the resulting XML message should be inserted into. Figure 2.2 visualizes this concept of queues and rules.

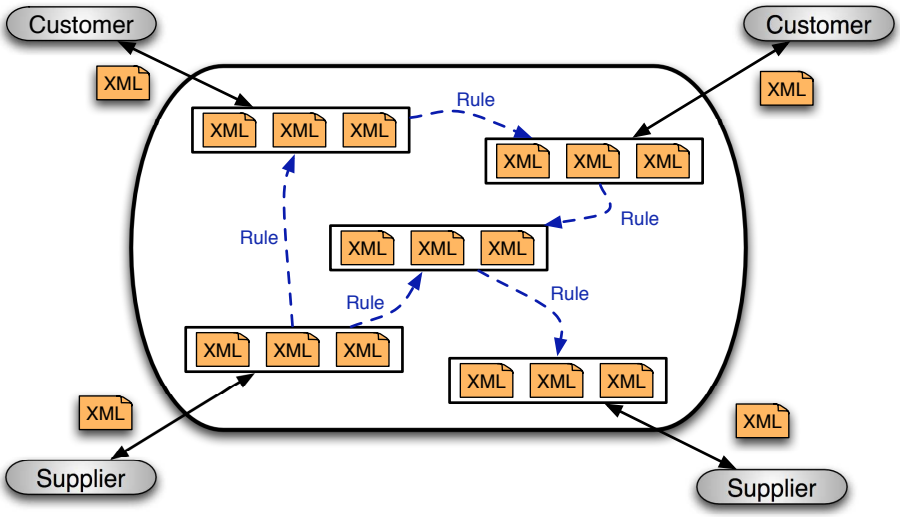

Figure 2.2: Demaq Programming Model

2.3 Differences

We now discuss the differences between the two different approaches introduced above. The main purpose of this section is to emphasize the fundamental differences in the programming principles of Demaq applications, compared to existing, imperative code.

2.3.1 Handling XML data

Most application servers treat the XML data send to and received from external systems as yet another aspect of the transport protocol. The XML data is hidden from the developer using a corresponding adapter, and made available as Java objects.

In Demaq, a developer directly operates on the XML data using the rules of the DQL programming language. The entire application is expressed by a set of rules querying XML data and producing new XML data.

2.3.2 Persistent Data Storage

Application servers achieve data persistence by using relational database management systems or object stores. Persistence management is either performed by the application server, a corresponding mapping framework, or directly e.g. using SQL/OQL code.

Demaq achieves data persistence by storing the complete *message history*. All messages received from and sent to external systems are stored persistently in the corresponding queues. Thus, queues are not only used as staging areas

for incoming and outgoing messages, but also serve as storage container. Application rules may access this data using corresponding functions (see Chapter 4).

2.3.3 Application Code

Most application servers use imperative programming programming languages (Java) to define the business logic. A developer provides the runtime system with a detailed execution plan, defining which steps should be executed whenever a particular event (e.g. a new message arrives) occurs. The potential for automatically code optimizations is rather limited, developers have to manually tune their application in order to achieve performance improvements. Debugging is rather easy, as the sequence of operations performed by the runtime system exactly corresponds to those defined by the developer.

In Demaq, an application is defined as a set of application rules, defined on the queues of the system. As DQL is declarative, the application specification is less detailed as for Java code, providing the system with a much greater potential to perform automatic optimizations, both with respect to optimizing single rules and with respect to the overall application (e.g. where two rules defined on the same queue can be merged).

As the operations performed by the runtime system may significantly differ from the specification initially provided by the developer, debugging a Demaq application is more complex than of e.g. Java code. Even worse, as the messages resulting from Demaq rule may be inserted into another queue, again having rules defined on it, there may be complex or even unexpected triggering relationships within a Demaq application. Similar to recursive trigger invocation in database management systems, these rule invocations are significantly more complex and non-trivial to understand.

2.3.4 Runtime Context and Instances

An application server usually allows for several concurrent program instances, each of them having its own context (e.g. represented using local variables). For example, the shopping cart of a particular instance is represented by a container class of items (e.g. a `set` in Java), and the shipping address may be represented by a simple string variable. To add an item to the cart, the corresponding set data structure can be modified in place by adding a new item to it. Similarly, if the user decides to change the shipping address, the corresponding string variable is modified in place in the context (e.g. using `address = newaddress;`).

In Demaq, there are **no runtime contexts** and **no in-place updates**. Particularly, the only means of storing and accessing data are message that are kept in queues. Thus, data handling is significantly different from e.g. Java code and (at least appears to be) more complex.

Acquiring context information

For each incoming message, there usually is a set of related messages that are required to process the business logic for the particular request represented by the incoming message. To access this set of related messages (e.g. all other messages retrieved for a particular customer, all current orders, etc.) a Demaq application can query the message history. For example, to retrieve all orders for a particular customer, an application rule could access the queue containing all orders and project on the relevant customer ID (e.g. using a path expression).

As access to such sets of related messages is a frequent operations, DQL allows to express typical access patterns using *slicings*, which will be introduced in Section 3.

Updating data

Apart from the lack of the runtime context for retrieving context data, Demaq does not provide any facilities for modifying data in place. While new messages may be added to queues, the content of existing messages may not be changed. Queues can be considered to be *append only*.

In the Java example above, the shipping address of a customer was easily changed by simply setting a variable in the runtime context to another value. In Demaq, this is not possible. Instead, the XML message reflecting the change request would be added to a queue of the system. When retrieving the shipping address, the corresponding application rule has to consider this change message in order to retrieve the correct address. This could, for example, be done by investigating all messages reflecting master data updates and picking the address from the last message containing a new address field.

Chapter 3

Queue Definition Language

Conceptually, the Demaq programming language can be divided into two parts. The *Queue Definition Language* is used to define the infrastructure of queues underlying any application. It is complemented by the *Queue Rule Language* which is used to define the rules implementing the actual application logic.

This section gives a detailed overview of the concepts and constructs of the Queue Definition Language (QDL). Section 3.1 starts with a description of the various kinds of queues that are supported. Afterwards, the concept of message properties is introduced in Section 3.2. Finally, Section 3.3 discusses how slicings can be used to define application-specific message contexts and to simplify application development.

3.1 Queues

Demaq exclusively relies on queues as the underlying data structures. Queues are used for two purposes: As asynchronous communication gateways to external systems and for persistent, local message storage. These two different tasks are reflected by two different queue *kinds*, that will be introduced below.

3.1.1 Gateway queues

In Demaq, *gateway queues* represent the connections to external systems. Messages that are received from external systems arrive at an (incoming) gateway queue, while messages that are placed into an outgoing queue are sent.

Thus, gateway queues are much similar to the queues offered by message-oriented middleware solutions, where they are used staging areas for messages that will be finally sent to external systems, or as message buffers that can be consumed by an application.

Examples

This example shows how to create gateway queues and the different parameters that are available.

Outgoing gateway queues The following expression creates a gateway queue with name `messageOutput` for outgoing data. Note that the name of a queue has to be unique, i.e. there must not be two queues with the same name.

```
create queue messageOutput kind outgoing mode persistent;
```

Incoming gateway queues The *kind* of a gateway queue definition defines whether the gateway queue is used for incoming or outgoing message. Thus, a queue `messageInput` for outgoing data would be defined as follows.

```
create queue messageInput kind incoming interface "example.wsdl"
    port "ExamplePort" mode persistent;
```

Apart from changing the `kind` to incoming here, we also need to define what kind of message are expected at the gateway queue, and what transport service should be used for the server listening for messages. The basic idea here is to specify a WSDL file that defines the corresponding interface and select a target port.

> Currently, there is no WSDL support in Demaq, thus the interface and port expressions are used to define the transport protocol in the interface part and the destination port for the transport protocol in the port part (e.g. interface "http" port "80" or interface "smtp" port "25"). Both parts can be left empty, in this case, no transport protocol will be used (but the communication channel can be accessed from the host C++ program running Demaq).

Synchronous transport protocols While messaging operations in Demaq are basically asynchronous, some transport protocols (e.g. HTTP) require synchronous data transfer, where the reply to a message is sent using the same connection as the initial request. To support this kind of interaction, gateway queues can be associated with a `response queue`. Messages that are placed into a response queue are automatically correlated with those from the associated gateway queue and sent using the same connection.

In the following example, we create an incoming gateway queue synMessages with a response queue synReplies.

```
create queue synMessages kind incoming interface "http" port "2342"
    response synReplies mode persistent;
```

The associated synReplies queue will be automatically created, thus manually creating another queue named synReplies is an error.

Using HTTP One of the supported, synchronous protocols is HTTP. In order to have the Demaq system listening for incoming HTTP requests on a particular port, a corresponding gateway queue has to be created. As there is no WSDL support yet, the interface definition statement is abused for this purpose.

```
create queue httpMessages kind incoming interface "http" port "2342"
    response httpReplies mode persistent;
```

The example above illustrates how to create gateway queue with name `httpMessages` listening on port 2342 for incoming HTTP requests. The `http` interface definition chooses the protocol to use, the port of `2342` indicates that the system should use this port for listening for incoming request. Replies sent to the `httpReplies` response queue will be automatically correlated with the initial, incoming request.

> It is not possible to sent multiple responses to a single incoming request. Every incoming request must be matched by exactly one response. Note that you *must* respond to the initial request, otherwise the system will not be able to receive subsequent requests over the gateway queue.

Using SMTP Another supported transport protocol is the asynchronous SMTP. Similar to HTTP, a corresponding gateway queue has to be created in order to retrieve SMTP requests.

```
create queue smtpMessages kind incoming interface "smtp" port "2342"
    mode persistent;
```

As SMTP is an asynchronous transport protocol, no response queue must be associated with the incoming gateway queue.

3.1.2 Basic queues

Basic queues are the storage containers for a Demaq application. Their simple purpose is to persistently store messages created by application rules without sending them to external systems. Basic queues particularly useful as intermediate storage buffers in order to materialize data, or to split complex application code into several, smaller rules that are executed one after the other, using basic queues for storing the intermediate result.

Example

The following example creates a basic queue with name storage.

```
create queue storage kind basic mode persistent;
```

3.1.3 Advanced features

Apart from the definition of basic and gateway queues as seen in the above examples, there are several additional options for queue definitions that will be

discussed below. Apart from the mandatory definition of the *mode*, all of these are optional and can be omitted from the queue definition.

Persistent and transient data storage

All queue definitions seen so far included the `mode persistent` expression which has not been discussed yet. Queues (both basic and gateway queues) can operate in two modes. A *persistent* mode indicates that all messages in a queue should be stored persistently and must not be lost in case of system errors, application crashes, error conditions, system shutdown, etc. These guarantees are not given for *transient* queues, where data may be lost in the cases listed above. Thus, transient queues involve less overhead and may provide faster data handling, but should only be used for those message that may be lost.

The example below creates a basic queue with name advertisement in transient mode.

```
create queue advertisement kind basic mode transient;
```

Priorities

Any queue may optionally be assigned a priority to indicate that the messages in a particular queue are considered to be more important that those in another one. Depending on the scheduling strategy used, these priorities affect the sequence in which messages are processed, e.g. messages in a queue with a higher priority may be processed before those in a lower priority queue, even when arriving later. Priorities are unsigned integer values, with a default value of 0.

The following example creates two queues with different priorities.

```
create queue important kind basic mode transient priority 23;
create queue moreImportant kind basic mode transient priority 42;
```

> Currently, the Demaq system does not consider priorities assigned to queues.

Schema validation

Optionally, any queue may be assigned a schema definition, restricting the type of messages that may be inserted into this queue. Any message that fails to match the schema definitions will cause a processing error (error handling will be discussed in Chapter 4.

> Schema validation is currently not implemented in the Demaq system.

Error queues

Whenever a message is being processed by an application rule, it may raise a runtime error. While the Demaq error handling strategy will be discussed

in Section 4.5, we already describe the QDL-specific part of error handling here. Basically, whenever an error is triggered by a message, a corresponding notification message is created and sent to the corresponding *errorqueue*.

To define which errorqueue is responsible for a message, an errorqueue may be defined for all messages in a particular queue. In the example below, two queues are created. The containsErrors queue is the errorqueue which will store the error notification messages. Whenever a message stored in the mayRaiseErrors queue triggers a runtime error, the corresponding notification message is sent to the associated errorqueue (containsErrors in this example).

```
create queue containsErrors kind basic mode persistent;
create queue mayRaiseErrors kind basic mode persistent
  errorqueue containsErrors;
```

> Any errorqueue must be defined by a corresponding QDL statement. Any kind of queue may be an errorqueue, e.g. a gateway queue may be an errorqueue.

3.2 Properties

Any message stored in the Demaq system may be annotated with additional *properties*. A property is a message-specific pair that associates a value with a unique key. In order to assign a property value within application rules (see Chapter 4), the property has to be previously defined. Properties are defined for all messages in a particular queue.

In the following example, we define a property isImportant for all messages in the incomingOrders queue. Note that the incomingOrders queue has to be previously defined.

```
create queue incomingOrders kind basic mode persistent;
create property isImportant queue incomingOrders;
```

For convenience, a single property definition may define a property for several queues. In the following example, the isImportant property is defined for both the incomingOrders and the outgoingOrders queue. For readability, we omit the queue definition statements in the following examples.

```
create property isImportant queue incomingOrders, outgoingOrders;
```

This definition could also be written as

```
create property isImportant queue incomingOrders queue outgoingOrders;
```

Optionally, properties may be assigned an XQuery type. In the example below, the isImportant property is defined to be of kind xs:boolean.

```
create property isImportant as xs:boolean
  queue incomingOrders queue outgoingOrders;
```

3.2.1 Computed properties

Instead of manually specifying a value for a property from application rules, properties can also be *computed* by the system. This is particularly useful if the value of the property already occurs in the associated message. For the isImportant property in the example above, this could e.g. be the case if the message optionally contains an `<isImportant/>` element.

For those cases, computed properties can be used to conveniently retrieve the corresponding information from the document. A computed property associates the property definition with a corresponding XQuery expression. Whenever the value of the property is accessed, this expression will be evaluated with the particular message as the context item.

In the example below, we will define the isImportant property as a computed property that will query the document for the presence of a corresponding isImportant element.

```
create property isImportant queue incomingOrders, outgoingOrders
  value //isImportant;
```

The only difference to the definition above is the new *value* part which associates the path expression to the property. Optionally, the same property may be defined with different value expressions for the queues it is defined on.

In the following example, a property customerID will be defined for three different queues. The messages in those queues have different schemas, thus the same customerID will be found in different parts of the corresponding messages, and might even have different names. For these cases, the property mechanism can be used to give an common name to those different customer handles and use a uniform mechanism to access the corresponding data.

```
create property customerID
  queue incomingOrders value /message/customer/ID
  queue internalProcessing value //customerID
  queue outgoingOrders value /reply/customerData/cID;
```

Note that properties may still be set manually from application rules. In this case, the value that has been set manually will be used instead of evaluating the query.

3.2.2 Fixed properties

As seen above, computed properties may be "overwritten" by manually setting the property to a particular value. Depending on the application, this behavior may not be desirable. Instead, in this cases the system should enforce that the property value will be computed, disallowing a manual specification.

For this purpose, the property may be defined to be *fixed*. Fixed properties may not be changed by application rules. In the example below, we add the fixed modifier to the above definition.

```
create property isImportant queue incomingOrders, outgoingOrders
```

fixed value //isImportant;

Note that the same property may be defined with a different modifier for each of the queue involved. For example, the customerID property above should only be fixed for the incomingOrders and internalProcessing queues, but not for the outgoingOrders queue. This is reflected by the following definition:

```
create property customerID
  queue incomingOrders fixed value /message/customer/ID
  queue internalProcessing fixed value //customerID
  queue outgoingOrders value /reply/customerData/cID;
```

3.2.3 Inherited properties

Once a message got annotated with a particular property (e.g. by setting it in an application rule), an application may want to propagate this property to the messages that are derived from it.

An example would be processing a high priority request from an important customer. In this case, not only the initial customer message should be annotated with a corresponding property (highPriority), but also all derived message should have the same property set in order to propagate the information throughout the entire Demaq application. Instead of manually setting the property within application rules, the property can be created with a corresponding *inherited* modifier, indicating that the value of the property should be propagated to all other messages derived from the message the property is defined on. Propagation is only performed when a property with the same name is defined on the queue the derived message will be inserted into.

In the example below, we create a highPriority property that will be inherited by the derived messages stored in the internalProcessing and outgoingOrders queue. For the incomingOrders queue, this property has to be set manually.

```
create property highPriority
  queue incomingOrders
  queue internalProcessing, outgoingOrders inherited;
```

To disallow manually overwriting inherited properties (e.g. to make sure that a derived message always has the same property value as the message it was derived from), the *fixed* modifier can be used as in the following example.

```
create property highPriority
  queue incomingOrders
  queue internalProcessing, outgoingOrders inherited fixed;
```

Note that defining a property as both inherited, fixed and computed is an error.

3.3 Slicings

The slicing mechanism is the most complex part of QDL, as it is based on both queues and properties. The motivation for the slicings mechanism is the observation that while queues can be used to group similar messages in a conjoint location in the message store, there are often multiple, orthogonal message groups that are meaningful for an application.

For example, an application may be interested in all orders (stored in a conjoint queue), but also in all customer transactions (containing orders, requests, replies, invoices, etc. arbitrarily distributed over the queues of the system), all messages sent from a particular customer, all high-priority messages, etc.

To allow application programs to easily access these logical groups of related messages without querying the content of the underlying queues and constructing a result set, the slicing mechanism can be used.

3.3.1 Using a slicing to partition a single queue

As an example, consider that a single queue contains the orders received by all customers of our online shop. However, in our application program, we only need to access those order on a per-customer basis, as any customer transaction only references the order of this particular customer. Thus, we will create a slicing that allows us to only retrieve these messages that are interesting in the particular customer context.

Any slicing definition is based on a (computed) property definition. Thus the first step is to define a (computed) property that allows us to distinguish between the different customers. For this purpose, we assume that very message contains an element with the particular, unique customerID (`<customerID>42</customerID>`).

The value of the property is this customerID, computed using a corresponding path expression.

```
create property customerID
  queue orders fixed value //customerID;
```

In the next step, the slicing is defined on the customerID property created above.

```
create slicing customerMessage on customerID require fn:false();
```

The resulting customerMessage slicing *partitions* the messages in the input queues (only the single orders queue in this example) according to the associated property value. For each distinct property value, a sequence of messages (*called slice*) is created, containing all the messages that share the same property value.

As a brief example, assume the orders queue contains five messages, each of them having a unique messageID and a customerID (denoted by a pair (messageID, customerID)). The content of the queue then looks as follows: (1, alex), (2, cc), (3, guido), (4, cc), (5, alex).

The slicing defined above would now partition all messages by the value of the property the slicing is defined on. In this example, there are three different

property values (alex, cc, guido), thus there will be three different slices. The slice for the property value (also called the *slicekey*) alex will contain the message sequence (1, alex), (5, alex), the slice with slicekey cc will contain (2, cc),(4, cc) and the slice with slicekey guido will contain (3, guido). Thus, as seen above, every slicing partitions the input queue into several, distinct slices, each of them containing those messages that share the same value for the property the slicing is defined on (aka slicekey).

> The *require* expression can be use to additionally restrict the messages that should be contained in the slice. A require expression of `fn:false()` indicates that no restrictions apply. The proper use of the require expression will be discussed in Section 3.3.4.

3.3.2 Using a slicing on several input queues

Apart from partitioning the input of a single queue into several slices as in the example above, slicings can also be used for groups of logically related messages that are distributed across several, different queues. An example for such a scenario in our online shop is to access *all* messages belonging to a particular customer, including requests, orders and confirmations, each of them stored in a corresponding queue of the system. Figure 3.1 depicts this scenario.

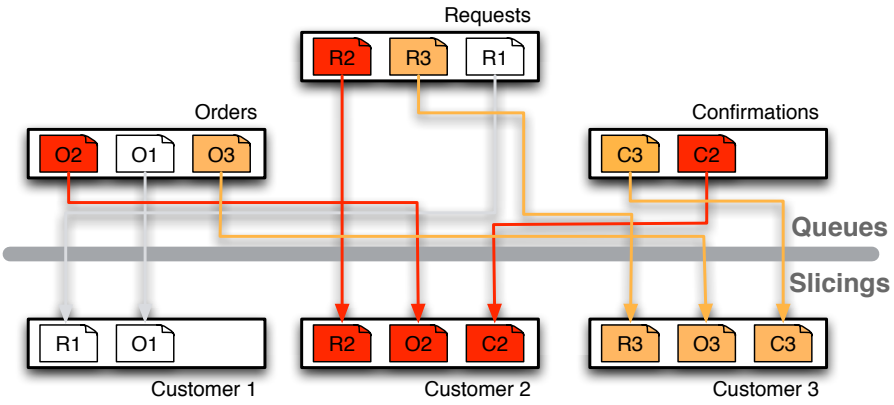

Figure 3.1: Using a slicing on multiple queues

The following example shows the corresponding QDL code (including the queue definitions):

```
create queue orders kind basic mode persistent;
create queue requests kind basic mode persistent;
create queue confirmations kind basic mode persistent;

create property customerID
  queue orders, requests fixed value //customerID
```

```
  queue confirmations fixed value /customer/cID;

create slicing customerMessages on customerID require fn:false();
```

As depicted by Figure 3.1, the slicing partitions the input from the underlying three queues using the customerID property. For each distinct value of the customerID property, a particular slice is created, containing all those messages from the underlying queues that share this property value. Note that the property is defined using different path expressions for the queues it is defined on, thus allowing to combine messages with different schemas into a conjoint slice.

3.3.3 Using slicings in application programs

As seen in the examples above, slicings can be used to access groups of logically related messages, independent of their queue storage location, thus facilitating retrieving all messages that are meaningful for application rules.

While there main purpose is to simplify application development, slicings also represent a efficient means of accessing messages stored in queues. For example, the Demaq system uses special index structures to speed up slice access, thus, the runtime performance of an application using slicings rather than directly accessing queues will be superior in almost any situation.

> Applications should favor using slicings to access messages over directly accessing queues.

Syntactical shortcuts

To speed up application development, there is a syntactical shortcut for slicing declaration. Apart from first creating a property and then a slicing on it, these two steps can be combined using the *create slicing property* statement as in the following example. This shortcut is particularly useful when a property is only used as the basis for a slicing definition and not required for other parts of the application code. The code below gives an example for such a combined declaration.

```
create slicing property customerMessage
  queue orders fixed value //customerID
  require fn:false();
```

This code is semantically equivalent to the following definition:

```
create property customerMessage
  queue orders fixed value //customerID;
create slicing customerMessage on customerMessage
  require fn:false();
```

3.3.4 Slice content visibility

The *require* expression part of a slice specification is used to restrict the messages that should be returned when accessing a slice. The motivation for the require constraint is that application programs often do not need to access the entire message history of a slice, but only parts of it. By choosing an appropriate require expression, the messages returned by the slice function can be restricted to this required part.

The require expression is a regular DQL/XQuery expression, however, some restrictions apply.

- The context item of the top-level expression is undefined and must not be accessed.
- The dynamic context is inherited from the rule calling the slice function, e.g. when using the `fn:current-dateTime` function within the require expression the current date/time of the context message of rule execution is used.
- An additional `qs:history` function can be used to access all messages in the slice history (all messages that have a matching slicekey).
- The `fn:collection, fn:document, qs:queue, qs:message, qs:slice` and `qs:slicekey` functions must not be used.
- The require expression must not perform update operations (e.g. `enqueue message`).

Result of the slice function

The slice function returns a *minimal valid suffix* of the input sequence. In the following, a slice containing k+1 message is considered to be a sequence of messages $[x_0, x_1, ..., x_k]$ (see Figure 3.2).

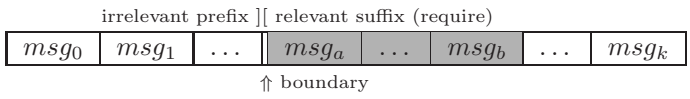

Figure 3.2: Message history in a slice of size k+1

Among all the contiguous sets of candidate messages in the slice that fulfill the require condition, the most recent set is considered the currently relevant state of the slice. This set (marked gray in Figure 3.2), and and any messages more recent than that, are visible to the application.

When computing the minimal valid suffix, there are two different cases that may occur.

1. There is a minimal valid suffix that fulfills the required condition. The slice function returns this suffix and all newer messages (msg_a to msg_k).

2. There is no sequence of messages in the entire input history fulfilling the require condition. In this case, the slice function returns all messages of the input sequence (msg_0 to msg_k). In this case, the result of the slice function are all messages returned by the `qs:history` function.

Examples

- Keep the last 50 messages of a slice

  ```
  create slicing property messagesByCustomer
    queue incoming fixed value //customerID
    require count(qs:history()/*) = 50;
  ```

- Keep messages for five years

  ```
  ... require exists(for $message in qs:history()
                return qs:property($message ,"timestamp")+xs:duration("P5Y")
                                    ge fn:current-dateTime())
  ```

- Keep all messages (of type x) forever

  ```
  require fn:false()
  ```

- Keep orders until a checkout arrives (from shopping cart)

  ```
  ... require count(qs:history()//checkout) eq 2
  ```

- Keep the last message of type x (shopping cart)

  ```
  require qs:history()//x
  ```

- Keep the last message

  ```
  require count(qs:history()) eq 1
  ```

- Keep same amount of orders and confirmations (at least one)

  ```
  require count(qs:history()//order) eq count(qs:history()//confirmation) and
          count(qs:history()//order) gt 1)
  ```

While Demaq conceptually provides access to the entire message history, the system applies garbage collection mechanisms to remove messages that are no longer required by application rule. The runtime system uses the criteria specified by the require expressions to decide whether or not a message is still required or can be safely deleted: Whenever a message is no longer contained in the result sequence of any slice it can be removed. Unprocessed messages are never removed, making sure that every message is considered by rule execution.
A require expression of `require fn:false()` indicated that the entire message history should be returned when accessing the corresponding slice. This is a very strict requirement, as no messages may ever be deleted from the system. When using this requirement, the system might eventually run out of storage capacity, thus this requirement should be avoided whenever possible.

Chapter 4

Queue Rule Language

As seen in the last chapter, the Queue Definition Language is used to create the infrastructure of queues, properties and slicings providing the foundation for any Demaq application. Within this chapter, we will discuss the Queue Rule Language (QRL), which is used to define the actual application logic as a set of declarative rules.

We first illustrate what rules look like and how they are used for message processing (Section 4.1). We then discuss how rules can access the content of queues, properties and slicings (Section 4.2). Afterwards, the `enqueue message` expression is described in depth, including its various, optional parameters. Finally, Section 4.5 illustrates how error handling is done in Demaq applications.

4.1 Overview

Any Demaq application consists of a set of rules governing the message flow between the underlying queues. For this purpose, every single rule defines how to react to a (single) message that is inserted into a specific queue of the system. Once a new message is inserted into this particular queue, the rule is executed, producing a (potentially empty) sequence of new XML messages as a result. These messages are then inserted into other queues of the system, potentially triggering other rules, or being transformed to a remote system when reaching a gateway queue (see Figure 4.1).

4.1.1 A first example: Hello, world

In this example, we illustrate how to create a simple hello-world application in Demaq. For this purpose, we need a gateway queue receiving a message from an external communication partner (and sending the response back using a response queue), as well as a single application rule.

```
create queue input kind incoming interface "" port "" response output
       mode persistent;
```

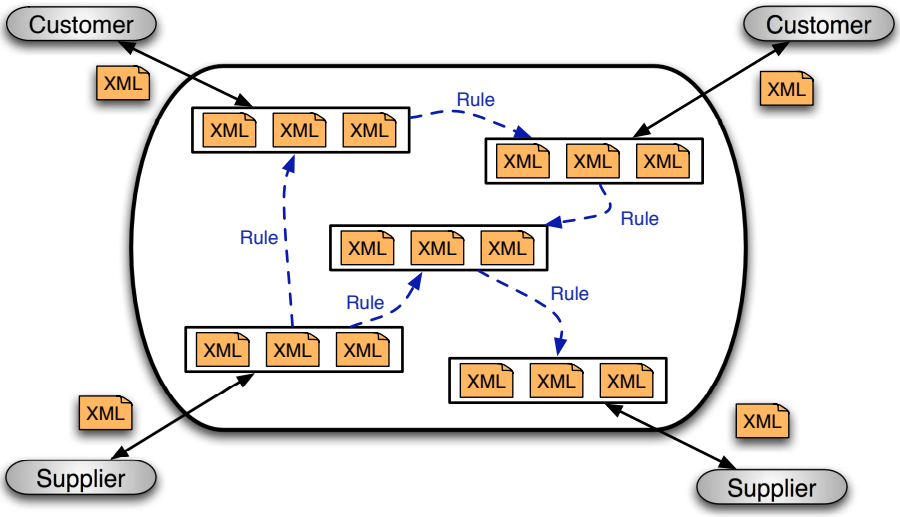

Figure 4.1: Demaq Programming Model

create rule helloWorld **for** input
enqueue message <greeting>Hello, world</greeting> **into** output;

Of course, the rule set of this example is very simple. It consists of a single rule, named `helloWorld`, that is defined on the `input` queue. Whenever a new message is being inserted into the `input` queue, this rule is evaluated for this particular message. In this example, the application code of the rule (aka the *rule body*) consists of a single `enqueue message` statement. It is used to enqueue an XML fragment into a particular queue of the system. Here, a simple XML message is inserted into the output queue (thus being sent as a response to the incoming request).

4.1.2 Writing rule bodies with XQuery

The QRL language is based on XQuery [1], a XML query language standardized by the W3C. Basically, the body of an application rule is an XQuery expression with some Demaq-specific extensions (e.g. enqueue message) that will be described later on (Sections 4.2 and 4.3). The fundamental difference is that instead of returning the result of the XQuery expression to a user, a rule enqueues the resulting XML fragments into other queues of the systems (using the enqueue message statement).

Thus, every Demaq rule consists of a *create rule* statement that is used to give a unique name to a XQuery fragment, and to associate it with a particular queue of the system. The following example illustrates the use of XQuery in another application rule (named doSomething). This rule is always executed when

a new message is inserted into the inputQueue, with that particular message as the context item (e.g. //foobar refers to all foobar elements in the message triggering rule execution).

```
create rule doSomething for inputQueue
  (:XQuery here:)
  let $x := //inputMessage/someThing
  for $y in $x
  where some $z in $y/order satisfies fn:count($z//item) gt 5
  return
    enqueue message $x into outputQueue (:demaq-specific:)
;
```

Demaq also inherits the prolog from XQuery. Thus, for example, user-defined XQuery functions may be specified in the prolog and used within application rules as in the following example, recreating the above hello-world application using an XQuery function.

```
declare function local:hello() {<greeting>Hello, world</greeting>};
create queue input kind incoming interface "" port "" response output
  mode persistent;

create rule helloWorld for input
  enqueue message local:hello() into output;
```

4.1.3 Rule definition

Several different rules may be defined on the same queue of the system. Whenever a message is inserted into this queue, all these rules are evaluated for every incoming message.

Apart from queues, rules may also be defined on slicings. These slicing rules will be evaluated whenever a new message is added to a particular slice (i.e. whenever a message is inserted into one of the queues the slicing property is defined on).

4.2 Data Access Functions

Apart from the most fundamental rules (such as the hello-world rule in the example above), rules usually have to access the messages stored in the queues and slices of the system to retrieve context information. For example, our online shop might want to check the number of previous orders for a particular customer requesting a discount. Consequentially, the application rule handling the discount request needs to access a corresponding slicing containing all customer messages, or access the queue containing all order messages.

For this purpose, Demaq incorporates several system functions that provide application rules with *read-only* access to the messages stored in the system.

All these functions are in the Demaq system namespace, which is bound to the prefix `qs` by default.

4.2.1 Retrieving the context item

The `qs:message()` function allows to explicitly access the message triggering the execution of a rule. This while this message is used as the context item for rule execution and can thus be (implicitly) accessed e.g. in path expressions, accessing this messages e.g. becomes necessary when the context item changes (e.g. in the predicate of a XQuery step expression).

In the example below, both variables x and y refer to the same item, the message triggering rule execution. The need for the qs:message() function becomes apparent when assigning a value to the z variable. Here, the context item in the predicate is the item1 element. Thus, in order to access the item2 element, the qs:message() function is used to explicitly access the triggering message and to subsequently retrieve the name element of item2.

```
create rule messageXS for someQueue
  let $x := .
  let $y := qs:message()
  let $z := /items/item1[name eq qs:message()/items/item2/name]
  return enqueue message $z into anotherQueue
;
```

4.2.2 Retrieving all messages in a queue

The `qs:queue("target")` function can be used to access all messages in a particular queue. It takes the name of the target queue as it's only parameter. The name of the target queue must be specified as a string (in particular it must not be an XQuery QName as the queue names usually are to meet the XQuery function call signature).

In the example below, the `qs:queue` function is used to retrieve all message from the orderMessages queue of the system. Afterwards, the number of contained messages is enqueued to another queue of the system.

```
create queue orderMessages kind basic mode persistent;
create queue anotherQueue kind basic mode persistent;

create rule countOrders for orderMessages
  let $orderMessages := qs:queue("orderMessages")
  return enqueue message
    <orderCount>{fn:count($orderMessages)}</orderCount>
    into anotherQueue;
```

Syntactical shortcuts

To simplify application development, the `qs:queue()` can be invoked without specifying the name of a particular queue as parameter. This shortcut may only be used in rules defined on queues and returns all messages contained in the queue the rule is defined on. In rules defined on slicings, this parameter has to be specified in any case.

In the example above, the rule is defined on the orderMessages queue, and the qs:queue function is used to retrieve the message from the same queue. Thus, the above rule is equivalent to the following one:

```
create queue orderMessages kind basic mode persistent;
create queue anotherQueue kind basic mode persistent;

create rule countOrders for orderMessages
  let $orderMessages := qs:queue()
  return enqueue message
    <orderCount>{fn:count($orderMessages)}</orderCount>
    into anotherQueue;
```

> Note that the qs:queue() function returns *all* messages stored in a queue of the system. Thus, qs:queue usually is a **VERY EXPENSIVE** operation and should only be used as a last resort. In almost any case, using a slicing is a much better alternative.

4.2.3 Retrieving the slicekey

As seen in Section 3.3, a slicing can be used to partition the messages from (several) input queues into groups of logically related messages. Each group of messages shares a conjoint value for the property the slicing is defined on, which is called the *slicekey*. In other words, all messages in the same slice have the same slicekey.

The `qs:slicekey("targetSlicing")` function can be used to retrieve the slicekey for the message triggering rule execution. The function takes the name of the target slicing as its single parameter.

```
create queue orders kind basic mode persistent;

create slicing property customerID
  queue orders fixed value //customerID

create rule uselessExample for customerID
  let $customerID := qs:slicekey("customerID")
  return
    if($customerID eq "42")
    then ... else ()
;
```

In the (simplified) example above, calling the `qs:slicekey` function is logically equivalent to directly using the `//customerID` path expression.

Syntactical shortcuts

In rules defined on slicings, the name of the target slice can be omitted if the slicekey of the current message should be retrieved with respect to the slicing the rule is defined on. In the example above, the `qs:slicekey("customerID")` function is semantically equivalent to simply writing `qs:slicekey()`, as the rule is defined on the customerID slicing.

> The qs:slicekey function may only be used in rules defined on *queues* if the property the slicing is defined on has a corresponding entry for this particular rule. Otherwise, as the messages of this queue are not part of the slicing, the slicekey function cannot be evaluated.

4.2.4 Retrieving the messages in a slice

The messages contained in a particular slice can be retrieved using the `qs:slice("slicekey", "slicename")` function. It takes two parameters: The first parameter identifies the key of the particular slice (the *slicekey*) which is the value that all messages in the particular slice share for the underlying property. The second parameter is the name of the slicing to which the particular slice belongs.

```
create queue orders kind basic mode persistent;
create slicing property customerID
  queue orders fixed value //customerID:

create rule doSomething on customerID
  let $interestingMessages := qs:slice("42","customerID")
  ...
;
```

In the example above, the `qs:slice` function is used to retrieve the slice with slicekey "42" from the customerID slicing. As the slicekey is defined by evaluating the `customerID` path expression on the corresponding messages (from the orders queue in this case), the result of the `qs:slice` function will contain all messages that have a customerID of 42.

In the example below, the `qs:slice` function is used to retrieve all messages that share the same slice key as the message triggering rule execution

```
...
create rule doSomething2 on customerID
  let $interestingMessages := qs:slice(qs:slicekey(), "customerID")
  ...
;
```

Here, the `qs:slicekey` function is used to dynamically retrieve the corresponding slice key from the triggering message.

Syntactical shortcuts

The `qs:slice` function may be invoked using several shortcuts. These shortcuts may only be used in rules defined on slicings.

The first shortcut is to invoke the `qs:slice` function with only a single parameter, defining the slicekey of the target slice to retrieve. In this case, the name of the target slicing defaults to the name of the slicing the rule is defined on. Thus, the following code is logically equivalent to the doSomething2 example above.

```
...
create rule doSomething2 on customerID
  let $interestingMessages := qs:slice(qs:slicekey())
  ...
;
```

As another shortcut, the `qs:slice` function may be invoked without any parameter. In this case, the slicekey defaults to the result of the qs:slicekey() function, while the target slicing name default to the name of the slicing the rule is define on. Thus, the example above can be abbreviated to the following code.

```
...
create rule doSomething2 on customerID
  let $interestingMessages := qs:slice()
  ...
;
```

4.2.5 Retrieving the value of a property

The purpose of the `qs:property("propertyName", $contextItem)` function is to retrieve the value of a particular property for a particular message. It takes two parameters, identifying the name of the property to retrieve the value for, and the particular message for which this should be done. The result of the property function may be empty, if the corresponding property value is not set for the corresponding message.

In the example below, the value of the property "customerSucks" is retrieved to the context item (the message triggering rule execution) accessed using the `qs:message` function.

```
create queue orders kind basic mode persistent;

create property customerSucks queue orders;

create rule handleCustomerOrder for orders
```

```
let $stupidCustomer := qs:property("customerSucks"), qs:message())
if($stupidCustomer) then ... else ()
...;
```

Syntactical shortcuts

In order to access a property of the message triggering the execution of the current rule, the contextItem parameter may be omitted. Thus, the rule in the example below is equivalent to that in the example above.

```
create rule handleCustomerOrder for orders
  let $stupidCustomer := qs:property("customerSucks")
  if($stupidCustomer) then ... else ()
  ...;
```

4.2.6 Retrieving the timestamp of a message

Every message has a timestamp that reflects the time it has been enqueued into the message store. The timestamp of a particular message can be retrieved by using the `qs:timestamp($message)` function, which will return the timestamp in `xs:dateTime` format.

The example below shows how to retrieve the timestamp of the first message in the `orders` queue.

```
create rule firstTS for someQueue
  let $message := qs:queue("orders")[1] (:first msg:)
  let $timestamp as xs:dateTime := qs:timestamp($message)
  ...
```

Syntactical shortcuts

Optionally, the target message parameter can be omitted when calling the `qs:timestamp()` function. In this case, the timestamp of the context message will be returned. Thus, calling `qs:timestamp()` is equivalent to calling `qs:timestamp(qs:message())`.

```
create rule contextTS for someQueue
  let $contextMessageTimestamp := qs:timestamp()
  ...
```

4.2.7 Retrieving the unique ID (messageID) of a message

In Demaq, every message has a unique messageID. More precisely, there will never be two messages in the context of a Demaq instance on a particular host that share the same messageID. MessageIDs are particularly helpful for uniquely identifying and accessing messages in application rules. For example,

messageIDs can be used to correlate an error message to the original message that triggered the error. As the messageID of a message does never change, using a messageID to reference a message is clearly superior to e.g. using the message's queue and a positional predicate (which may change due to subsequent enqueues or deletes triggered by the garbage collector).

The unique ID of a particular message can be accessed using the `qs:messageID($message)` function. In the example below, it is used to return the messageIDs of all messages in the `orders` queue.

```
create rule determineMsgIDs for orders
  let $messages := qs:queue()
  let $messageIDs :=
    for $message in $messages
    return qs:messageID($message)
  return ...
```

Syntactical shortcuts

When invoked without a parameter, the `qs:messageID()` function returns the unique ID of the context message. Thus, calling `qs:messageID()` is equivalent to calling `qs:messageID(qs:message())`.

```
create rule contextMsgID for someQueue
  let $messageID as xs:string := qs:messageID()
  return ...
```

4.2.8 Creating a unique identifier within a rule

There are several situations where an application might need to create a unique identifier, e.g. when creating master data that should be accessed by a unique ID later on. For this purpose, Demaq incorporates the `qs:uniqueID()` method. When called within an application rule, the method yields a string identifier that is unique in the context of a Demaq instance.

Calling the `qs:uniqueID()` function multiple times within a single rule yields the same result on each invocation. Calling the function in different rules for the same context item also yields the same result on each invocation.

In the example below, the function is used to create an identifier for a new customer record.

```
create rule addCustomer for newCustomers
  let $result as node() := <customer>
    <customerID>{qs:uniqueID()}</customerID>
    <data>{//customerData}</data>
  </customer>
  return $result into customers;
```

The `qs:uniqueID` method *can not* be used to create more than a single unique ID within a single rule, or within different rules evaluated on the same context message. Thus, to create multiple unique identifiers, additional intermediate queues have to be used.

4.3 Enqueue message expression

While all the system provided functions discussed in the previous sections only allow *read-mode* access to the underlying message store, the `enqueue message` statement can be used to actually modify the content of the messages in the queues of the system.

Particularly, the `enqueue message` expression is the *only* way to perform modifications to the message store, and it only allows for *append* operations, i.e. adding new messages to an existing queue.

4.3.1 Enqueuing XML fragments into a queue

The enqueue message expression was already used in some of the examples in this section to add an XML fragment to a particular queue. The following example again illustrates how this is done.

```
create queue orders kind basic mode persistent;
create queue confirmations kind basic mode persistent;

create rule handleOrders for orders
  let $requestedItems := //items
  let $confirmation := <order>{$requestedItems}</order>
  return enqueue message $confirmation into confirmations
;
```

The enqueue message statement also allows to insert the same message into several queues at the same time. For this purpose, the target expression (after the `into` expression) may be an XQuery enclosed expression, returning a list of target queue names as strings. In the example below, a message is inserted into two queues.

```
create rule handleOrders for orders
  let $requestedItems := //items
  let $confirmation := <order>{$requestedItems}</order>
  return enqueue message $confirmation into {"confirmations", "orders"}
;
```

Note that the enqueue message statement may only insert messages into *queues*. Particularly, the target of an enqueue statement must not be a slicing.

4.3.2 Defining message properties

When enqueuing a message, the (non-fixed) parameters defined for this message (see Section 3.2) may be explicitly set by an application rule. For this purpose, an enqueue message expression may be used with any number of optional property declarations. In the example below, two property values are set for the XML fragment inserted into the orders queue.

```
create queue orders kind basic mode persistent;
create queue confirmations kind basic mode persistent;

create property customerSucks as xs:boolean
  queue confirmations;
create property targetAddress as xs:string
  queue confirmations;

create rule handleOrders for orders
  let $requestedItems := qs:message()//items (: same as above:)
  let $confirmation := <order>{$requestedItems}</order>
  return enqueue message $confirmation
    into confirmations with customerSucks value fn:false()
      with targetAddress value //targetAddress/text()
;
```

> Parameter values that are declared as fixed may not be explicitly specified using a with-value expression.

4.3.3 Performing delayed message enqueuing

Depending on the business logic being implemented, an application may need to consider temporal aspects. For example, it may only make sense to send payment reminder message to a customer two weeks after the initial invoice has been sent. In Demaq, all time-related events and aspects are represented by notification messages.

For this purpose, an optional at-expression can be added enqueue message expression. It specifies a timestamp when the message should be enqueued into the target queue (the default is immediately). The temporal parameter for the at expression has to be in the XQuery time format. In the example below, the enqueuing of a message is delayed by two days.

```
create rule delayMessage for orders
  enqueue message <foo/> into confirmations
    at fn:current–dateTime()+xs:dayTimeDuration("PT2D");
```

In this example, the current timestamp is retrieved using an XQuery function, before adding a duration of two days to it to realize the delay.

There is no possibility to cancel a message once it has been scheduled for later delivery. Thus, there are only few cases where directly sending a message to an external system using the `at` parameter does make sense. Instead, most applications may want to use the delayed enqueuing to realize some sort of temporal callback mechanism.

4.3.4 Sending messages over a gateway queue

An XML message can be easily sent to a remote system by enqueuing it into a gateway queue. Currently, the two transport protocol available to interact with remote systems are HTTP and SMTP.

Sending a message using HTTP

In order to send a message via HTTP a corresponding, outgoing gateway queue has to be defined in the first place.

To define the destination URL, method (get or post) and port, the system expects special properties.

```
create queue outgoingGateway kind outgoing interface "http" port "2342"
    response reply mode persistent;
create queue someQueue kind basic mode persistent;

create rule exampleRule for someQueue
enqueue message . into outgoingGateway
   with comm:URL value "http://www.demaq.net/example"
   with comm:TransportProtocol value "comm:HttpGet"
   with comm:DestinationPort value "80";
```

As depicted by the above example, the `comm:URL` property defines the target URL to send a message to. `comm:TransportProtocol` identifies the transport protocol to use, currently these are `comm:HttpGet` or `comm:HttpPost`. Finally, `comm:DestinationPort` identifies the target port (the default is "80" if not set explicitly). Thus, the exampleRule above would forward any message to http://www.demaq.net/example using HTTP GET and port 80.

Note that the queue definition statement (create queue) has to include the transport protocol to use (e.g. HTTP) in the example above. If the transport protocol is omitted, the system is currently not able to start up the corresponding communication channels. Thus, sending messages will not work in this case. Apart from the protocol to use, all parameters such as destination URI, port and transfer method (e.g. get/post) can be defined in the enqueue statement as in the example above.

Sending a message using SMTP

Sending a message using SMTP is much similar to using HTTP, however, other transport protocol parameters have to be defined. Again, a corresponding gate-

way queue has to be create in the first place, and system properties are used to define the transport protocol parameters.

```
create queue input kind basic mode persistent;
create queue output kind outgoing interface "smtp" port "25" mode persistent;

(:simply forward message:)
create rule forward for input
(
  enqueue message . into output
        with comm:URL value "pi3.informatik.uni-mannheim.de"
        with comm:DestinationPort value "25"
        with comm:From value "alex@demaq.net"
        with comm:To value "alex@pi3.informatik.uni-mannheim.de"
        with comm:Subject value "Demaq test message"
)
;
```

Five system properties are meaningful for sending messages over SMTP. The `comm:URL` identifies the mail server to be used for sending the message, `comm:DestinationPort` identifies the port the mail server is using for handling SMTP requests (this parameter is optional, defaulting to port 25). The sender and recipient addresses are defined using `comm:From` and `comm:To`, respectively. Optionally, `comm:Subject` sets a particular subject for the outgoing message. These system can also be used to inspect metadata of incoming SMTP requests (see Section 4.3.6).

4.3.5 Handling incoming HTTP GET requests

Demaq supports incoming HTTP GET requests. As these requests (in contrast to post) do not include any message payload, the system automatically creates a single dummy element as corresponding message. The parameters of the incoming GET request (e.g. the local part of the destination URL) can be accessed as system properties using the `qs:property` function.

The following example shows how to react to an incoming request by sending back a HTML page, including the requested local path (retrieved using the URL system property).

> Note that the `comm:Encoding` property is explicitly set to `comm:HTML` in order to allow the target browser to properly interpret the result page as XHTML (instead of displaying the XML document in plain text format).

```
create queue httpin kind incoming interface "http" port "2342"
    response httpout mode persistent;

create rule machwas for httpin
let $result := <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
lang="de"><head><title>Test</title></head><body><h1>Demaq Testseite</h1>
```

```
<p>Requested path was: {qs:property("comm:URL")}</p></body></html>
return enqueue message $result into httpout
  with comm:Encoding value "comm:HTML"
;
```

4.3.6 System-provided properties for gateway queues

For *incoming gateway queues*, the system provides the following properties that can be accessed by application rules. The values of these properties can be retrieved using the `qs:property` function (see Section 4.2.5).

- `comm:URL` representing the local path of the target URL (e.g. `/foobar/bumm`) (*HTTP only*).
- `comm:Header` containing the complete client request (e.g. `POST /foobar/bumm HTTP/1.0`) (*HTTP only*).
- `comm:TransportProtocol` the transport protocol that has been used (either `comm:HttpPost`, `comm:HttpGet` or `comm:Smtp`).
- `comm:CorrelationID` a correlation identifier that can be used to indicate that a response message should be sent as a reply to this particular message. This property is only meaningful when using synchronous transport protocols. See section 4.3.7 for and example and a detailed explanation how to use the correlation ID.
- `comm:From` The sender of a message transferred using SMTP (*SMTP only*).
- `comm:To` The recipient of a message transferred using SMTP (*SMTP only*).
- `comm:Subject` The subject of a message transferred using SMTP (*SMTP only*).

For *outgoing gateway queues*, the following properties can be defined.

- `comm:URL` represents that complete target URL a message should be sent to (e.g. `http://www.foobar.invalid/bumm/baz` (*HTTP*) or the destination mail server to use for sending this message (*SMTP*).
- `comm:TransportProtocol` the transport protocol that has been used (either `comm:HttpPost`, `comm:HttpGet` or `comm:Smtp`)
- `comm:DestinationPort` the port the message should be sent to (e.g. `80` for HTTP in most cases, `25` for SMTP).
- `comm:Encoding` The transport encoding that should be set for the transport protocol. Available choices are `comm:XML` (default) and `comm:HTML`. In order to allow proper displaying of XHTML pages in browsers when (ab)using Demaq as a web server, `comm:HTML` has to be used (*currently HTTP only*).

- `comm:From` The sender of a message transferred using SMTP (*SMTP only*).
- `comm:To` The recipient of a message transferred using SMTP (*SMTP only*).
- `comm:Subject` The subject of a message transferred using SMTP (*SMTP only*).

4.3.7 Correlating reply messages to incoming requests

When using synchronous transfer protocols, each incoming message has to eventually result in an reply message. This reply is sent by enqueuing it into the response queue associated to the incoming gateway queue that received the message (see section 3.1.1 for details on gateway queues).

The system automatically tracks which messages are created as a consequence of an incoming message and recursively propagates this information (as a inherited system property). Thus, whenever the reply message is generated as a consequence of the incoming message, the system automatically infers which connection the reply has to be sent over. However, when the initial message and the reply are decoupled (e.g. the reply is triggered by a message arriving from another external system as in the example below), no automatic message correlation can be performed.

In this case, application developers need to provide the system with the necessary information by manually setting the `comm:CorrelationID` when enqueuing the reply message into the response queue.

The following example illustrates how this can be done. Here, a reply to a message arriving on the upstreamInput queue is sent whenever a request arrives on the downstreamInput queue. In order to identify the upstream message to which the response belongs to, the corresponding rule retrieves the correlation ID of the upstream message. Then, the reply is enqueued in the upstream response queue. By setting the correlationID explicitly, the system is able to identify the connection the message should be sent to as a reply.

For sending the downstream reply, the system is able to automatically infer the corresponding connection. This is possible as the result message (< *done*/ >) is created as a consequence of rule execution on the corresponding incoming message. Thus, the correlation ID can be automatically inferred in this case and does not need to be given explicitly.

```
create queue upstreamInput kind incoming interface "http" port "2342"
  response upstreamOutput mode persistent;
create queue downstreamInput kind incoming interface "http" port "4223"
  response downstreamOutput mode persistent;

create rule downStream for downstreamInput
let $correlationID:=qs:property(qs:queue("upstreamInput")[position() eq 1],
  "comm:CorrelationID")
return
```

```
(
  enqueue message . into upstreamOutput
    with comm:CorrelationID value $correlationID,
  enqueue message <done/> into downstreamOutput
);
```

4.4 Additional Demaq Updating Expressions

In addition to the `enqueue message` expression, Demaq incorporates other updating expression that can be used to control various aspects of the runtime system. These expressions are discussed in the following subsections.

4.4.1 Triggering System Shutdown

Demaq incorporates the `shutdown system` expression that can be used to shut down an active Demaq instance from an application rule. In the example below, system shutdown is triggered when a message with an `shutdown` element is enqueued in the `exampleQueue` queue.

```
create queue exampleQueue kind basic mode persistent;

create rule handleShutdown for exampleQueue
if(//shutdown)
then
  shutdown system
else
();
```

4.4.2 Tracking System Activity

By using the `request idle notification` expression, application developers can request the runtime system to send a (single) notification message to the system queue (`demaq:systemMessages`) whenever there is no further work to do. After a `request idle notification` expression has been called, the Demaq runtime system inspects the internal schedulers and sends the notification (containing a single `<systemIdle/>` element) when no more messages are scheduled. This expression can be e.g. used to shutdown the system when there is no more work to do, or to perform application-dependent cleanup operations in periods without application load.

In the example below, a idle notification is requested whenever a message with a particular element arrives at the example queue. If the system has no more work to do, the corresponding notification is sent to the demaq:systemMessages queue. In the example, this notification triggers a complete system shutdown.

```
create queue exampleQueue kind basic mode persistent;

create rule requestIdleNotification for exampleQueue
if(//requestIdleNotification)
then
request idle notification
else
();

create rule shutdownWhenIdle for demaq:systemMessages
if(//systemIdle/)
then
shutdown system
else
();
```

4.4.3 Requesting Garbage Collection

The `request garbage collection` expression allows to invoke Demaq's message and slicing garbage collection mechanism (see Section 5.2.1 for a detailed discussion) from application rules. This expression is particularly useful to trigger garbage collection in periods of low load (as e.g. indicated by the idle notifications discussed above). Explicit invocation of the garbage collector is also possible if automatic garbage collection has been turned of during instance creation (see Section 5.1.1).

The example below invokes the garbage collector each time a message is enqueued to the invokeGC queue.

```
create rule exampleRule for invokeGC
  request garbage collection;
```

> While garbage collection allows to reclaim storage space, garbage collection is a complex operation. Depending on the structure of the application, garbage collection may be a **VERY EXPENSIVE** and should be invoked carefully.

4.5 Error Handling

A Demaq application incorporates several sources of errors. This includes network-related problems such as disconnected transport endpoints, runtime errors caused by invalid XML fragments and broken application rules, as well as system-related failures, e.g. insufficient storage capacity or implementation errors. To allow applications to deal with these errors, Demaq translates all runtime errors to notification messages that are inserted into corresponding error queues.

By defining rules on these error queues, application developers may use QML to implement the corresponding error handling code.

4.5.1 Default error queue

For each Demaq application, a default error queue may be declared in the XQuery prolog. Unless any more specific error handlers (which will be discussed in the following sections) are in place, all error messages are inserted into the default error queue.

```
declare default errorqueue myErrorQueue;

create queue myErrorQueue kind basic mode persistent;

create rule myErrorHandler for myErrorQueue
...;
```

The specification of a default error queue is optional, by default, the Demaq system messages queue (demaq:systemMessages) will be used for this purpose.

> Note that the queue selected as the default errorqueue must be created using a corresponding DQL `create queue` expression. You may use any kind of queue as the default error queue (e.g. also a gateway queue to propagate errors to external systems).

4.5.2 Queue-specific error handlers

More specific than the default error queue, an error queue may be defined for all messages stored in a particular queue (as seen in Section 3.1.3). If a processing error is encountered for a message in this particular queue, the notification message will be sent to the errorqueue associated to the queue, instead of propagating it to the default error queue.

In the example below, any error message reflecting a processing error of a message in the orders queue will be enqueued to the orderErrors queue.

```
create queue orders kind basic mode persistent errorqueue orderErrors;
```

4.5.3 Rule-specific error handlers

Even more specific than queue-specific error handlers are rule-specific error queue. Whenever an error is encountered during processing of a rule with an associated error queue, the error message is inserted into this error queue. In the example below, all errors encountered when processing the handleOrders rule will be reflected by error messages sent to the processingErrors queue.

```
create queue orders kind basic mode persistent;
create queue processingErrors kind basic mode transient;
```

```
create rule handleOrders for orders errorqueue processingErrors
  ...
;
```

4.5.4 Error queue selection

In order to find the errorqueue appropriate for a particular error, the runtime system always picks the most selective errorqueue definition.

1. **Rule**: If an error is encountered during rule execution, the error message will be inserted into the errorqueue associated to the rule.
2. **Queue**: If there is no rule-specific errorqueue (or the error was not raised in the context of a rule), the errorqueue of the queue the message raising the error is stored in is chosen.
3. **Default error queue**: If there are neither rule- nor queue-specific error queues defined, the error notification messages is enqueued into the default error queue.
4. **Demaq system queue**: If there is even no default error queue defined, the error message is inserted into the Demaq system queue as the last resort. This queue contains all system-related notification messages (e.g. application startup and shutdown). The name of this queue is demaq:systemMessages.

```
declare default errorqueue genericErrors;

(:error handling queues:)
create queue ruleExecutionErrors kind basic mode transient;
create queue queuebasedErrors kind basic mode transient;
create queue genericErrors kind basic mode transient;

(:application logic queues:)
create queue orders kind basic mode persistent errorqueue queuebasedErrors;
create queue confirmations kind basic mode persistent;

create rule handleOrders for orders errorqueue ruleExecutionErrors
  let $x := ...
;

create rule countOrders for orders
  let $y := ...
;

create rule readConfirmations for confirmations
  let $z := ...
;
```

In the example above, errors encountered for the `handleOrders` rule will be handled by the ruleExecutionErrors queue, problems during execution of the `countOrders` rule will be inserted into the queuebasedErrors rule (as it has been assigned to the queue the rule is defined on), while errors for the `readConfirmations` rule are handled by the genericErrors queue.

4.5.5 Error Message Format

All error messages adhere to a conjoint schema. Within this section, we discuss the individual parts of error messages and their meaning. The detailed definition of the error message format can be found in the corresponding schema document in the Demaq code base (`docs/errormessage.xsd`).

The following example shows an error message that is created when encountering missing transport protocol parameters (e.g. not comm:URL property given when using HTTP as transport protocol).

```
<?xml version="1.0" encoding="UTF-8"?>
<error>
  <missingTransportProtocolParameters/>
  <diagnosis>Could not send message.</diagnosis>
  <description>Missing transport protocol parameters.</description>
  <context>
    <queue>output</queue>
  </context>
</error>
```

Each error message has an error element as its root element. Next, an element indicates what kind of error has occurred (a list of possible error kinds can be found below). The `diagnosis` part gives a brief diagnosis about the error. A detailed `description` can be found in the corresponding element. The optional `context` describes the application context in which the error occurred. In the example above, the error occurred when trying to send a message from the `output` queue. Below, we discuss all possible elements that may appear in under the `context` element.

Error Kinds

- `malformedXML` The system tried to enqueue a malformed XML message into the message store. This error could either result from application rules creating malformed message, or when retrieving a broken input document from an external system using a gateway queue.

- `ruleExecutionError` The system encountered an error during rule execution. This can, for example, be due to a dynamic error in XQuery evaluation, or result from difficulties in accessing and interacting with the underlying message store.

- `disconnectedTransportEndpoint` A remote transport endpoint unexpectedly terminated an ongoing data exchange (e.g. by a premature disconnect).
- `missingTransportProtocolParameters` A message enqueued into a gateway queue can not be send to external systems due to missing protocol parameters (such in the example above). See Section 4.3.6 for a list of required/supported properties that can be used to define transport protocol parameters.

Context Elements

Several elements are included (if applicable) in the error message to allow for detailed tracking of the context in which a particular error occurred.

- `rule` The rule that has been processed when the error occurred (if any).
- `queue` The queue a message is stored or should be stored (if applicable).
- `messageID` The messageID of the message being handled (if any).
- `message` A CDATA-included version of the message being processed.

4.6 Processing model

After the syntax of the QML elements and their semantics have been introduced, this section gives an overview when and how rules are evaluated, how results are produced and when they are incorporated into the message store.

Currently, Demaq uses a very basic FIFO scheduling strategy. This means that if a message is enqueued into any queue of the system before another message, it will be processed first. This means that there is a total timestamp-based order of messages, and messages will be processed sequentially. This scheduling policy is very primitive (e.g. it does not consider queue priorities) and is likely to change in future versions of the runtime system.

Once a message is being processed, all the rules that refer to it (i.e. all rules defined on the queue the message is stored in and defined on all slicings this queue participates) are evaluated, virtually at the same time. The result of rule execution is a list of resulting messages that have to be added to queues of the system. If the execution of a particular rule raises an error, an error message reflecting the problem is inserted into the associated error queue. Other rules potentially defined for this message remain unaffected and are processed regularly.

After rule processing, the resulting (regular and error) messages are incorporated into the message store, and the scheduler picks the next message to process.

Note that there is no ordering between the messages resulting from rule execution. Thus, if rule execution of a particular message result in two messages A and B being inserted into the message store, it is undefined whether A or B are processed first. This indeterminism arises as A and B have the same creation timestamp. If your application is sensitive to this kind of message ordering, you should avoid executing the rule creating A and those creating B at the same time (e.g. by introducing an intermediate queue). This indeterminism does not affect rules creating multiple messages as the rule-specific result set is a message *sequence*.

4.7 Application Modularization

The QRL language incorporates the concept of *application modules*. Using this concept, logically independent parts of an application can be implemented separate from each other in the form of individual, mutually independent modules.

Application modules allow developers to factorize reoccurring tasks that have to be performed multiple times in an application without duplicating the corresponding implementation, and to share reoccurring code among different application programs. Additionally, building a application out of several, independent modules instead of having a single, large code base helps to simplify development with multiple participants as each developer may individually work on a subset of the modules of the application.

For similar purposes, XQuery incorporates the concept of *library modules*, that allow to build function libraries that may be imported and subsequently used by other XQuery expressions. As these library modules do not incorporate support for the specific constructs of our programming language, such as queues, rules and slicings, they can only be used to factorize some application aspects such as reoccurring functions, but fail to support the factorization of complex application parts in general [7].

To mitigate this limitation, our application language incorporates a module concept, orthogonal to the XQuery library modules, which incorporates all constructs of our programming language, thus allowing to factorize individual, logical application parts to corresponding modules. In the following subsections, we discuss the key design aspects of our modularization concept, and show how it can be used to specify individual modules and to import and instantiate modules in the context of an application.

4.7.1 Module Design

To implement application modules, the complete functionality of QRL can be used. This allows application developers to factorize existing code as well as to separately implement new applications with the help of modules, and, in particular without any functional or syntactical limitations.

Apart from facilitating application development, an important goal of the design of the application modules was to strictly separate the application logic

provided by the module and that of the importing application. In particular, these two components should be loosely coupled and make as few assumptions as possible about each other to allow for independent development and module reuse. To achieve loose coupling, the only way of exchanging data between a module and the importing application is by means of message passing. For this purpose, each module provides an interface definition consisting of a set of incoming and outgoing queues. Much similar to the gateway queues that are used for the interaction with external systems, the incoming queues of a module used to receive messages sent by the importing application, while outgoing queues allow the module to send messages to the importing application.

Apart from sending and receiving messages using these interface queues, no data access and messaging operations between the module and the importing application are allowed. This includes data access functions (e.g. `qs:slice`) that may not be used to access messages of the module from within the importing application or vice versa, as well as using the `enqueue message` update operation to change the content of a queue of the importing application from a module (or vice versa) and may thus trigger data flow unexpected by the developer of the importing application.

In summary, from the point of view of an importing application, modules are 'black boxes' that provide their services over well-defined, queue-based interfaces and may never directly access or update data in the queues of the importing application. Consequentially, using this concept, application modules and functionality provided by external web services are handled using similar queue, based mechanisms.

4.7.2 Application Module Specification

In essence, a module specification is a full-fledged application program that may use the entire functionality provided by our programming language. This includes infrastructure definitions of queues, properties and slicings, application rules that reflect the logic to be implemented by the module as well as the import of other application modules (which will be discussed below). The main difference between an application module and a stand-alone application lies in the interface definition expressions. While the interface of a stand-alone application consists of the incoming and outgoing gateway queues that are used for the interaction with *external* systems, a application module specification includes a set of specific queues that are used as the message-passing interface for the communication with its importing application. In this interface definition, which includes both the queues that are used to receive messages from the importing application, as well as outgoing queues used to send messages, are made explicit. This explicit interface declaration enables developers to directly understand the interface of a module. Additionally, it allows for the automatic verification of an module import by the language, which may verify that all queues in the module interface have been properly assigned to queues of the importing application.

Example: Module Specification In the first two lines, the `declare input queue` and `declare output queue` expressions are used to define the incoming and outgoing interface queues of the module. In the example, these two queues are used by the module to exchange messages with the importing application. In the remainder of the module specification, regular language constructs such as slicings and rules are used to express the application logic of the module. A rule (line 9ff) reacts to all messages received from the importing application by enqueuing messages into the `outgoingMessages` queue. This queue represents the outgoing interface of the application module, which is used to send messages back to the main application.

```
1  declare input queue incomingMessages kind basic mode persistent;
2  declare output queue outgoingMessages kind basic mode persistent;
3
4  (: must retain all messages :)
5  create slicing property ICElog
6    queue incomingMessages fixed value //SHIPPING_O_ID/text()
7    require false();
8
9  create rule handleExternalICE for incomingMessages
10 let $response := <ICE_ReorderResponse>
11     <ICE_ReorderResult>OK</ICE_ReorderResult>
12     {//SHIPPING_O_ID} (:added for correlation to initial request:)
13   </ICE_ReorderResponse>
14 return enqueue message $response into outgoingMessages;
```

4.7.3 Module Import and Instantiation

An application module may be used by both stand-alone application as well as other application modules. In order to use a module, two steps need to be performed. First, the module has to be *imported* into the application. In this step, a corresponding import expression is used to assign a unique name to an existing module at a particular location (identified by a uniform resource identifier (URI)). This name can then be used to refer to the module in the remainder of the application.

After the module has been imported, it can be instantiated by the application using a corresponding *module binding*. A module binding creates an instance of a module and defines a map from a set of queues in the application to the queues in the interface definition of the module. Consequentially, the binding expression defines which queues are used by the application to send data to the module instance, and the queues that are used to receive messages in a response.

Example: Module Import and Instantiation In this example, we illustrate how to import and instantiate a module. First, the `import dqlModule` expression in line 1 is used to give a name (`externalICE`) to the module that can be found in a file identified by the corresponding URI.

Next, the `create binding` expression (line 3 ff) creates an instance (named `inventoryControl`) for the module previously imported. The `assign input` expression is used to map the `inventoryInput` queue of the importing application to the `incomingMessages` queue of the module. As a result, all messages enqueued into the `inventoryInput` queue will be passed to the corresponding queue in the module. Similarly, the `assign output` queue maps the outgoing interface queue of the module to a queue in the importing application.

```
1 import dqlModule externalICE at "./tpc_app/modularized/ice.dql";
2
3 create binding inventoryControl for externalICE
4   assign input inventoryInput as incomingMessages
5   assign output inventoryOutput as outgoingMessages
6 ;
```

It is important to note that while a binding expression defines a map of queues in the importing application to the queues of the module, the corresponding queues are separate from each other, an messages are sent from one queue to the other. Thus, in the example above, any messages that the module enqueues into its `incomingMessages` queue would never trigger the execution of a rule defined on the `inventoryInput` queue of the importing application.

In addition to the main concepts and functionality provided by application modules discussed above, modules may optionally incorporate *module parameters*. These parameters can be used to assign a value for parts of the module that depend on the importing application, such as port numbers, URIs or other, application-specific information. We refer the interested reader to [7] for a detailed discussion of these module parameters, together with additional application examples, an in-depth exploration of the design space as well as a brief performance evaluation.

4.8 Debugging applications

Compared to applications written in imperative languages (such as C++ or Java), debugging a Demaq application is significantly more complex. This mainly results from using a declarative language for the specification of the application logic. The Demaq language and runtime system provide several features to simplify debugging and understanding runtime behavior. These facilities will be discussed in the following sections.

4.8.1 Calling trace methods in application rules

To simplify application debugging, Demaq provides two different trace functions that can be embedded into application rules. These functions, called `qs:trace` and `qs:traceMessage` can be used to write messages or XML data to the system console.

qs:trace The `qs:trace` function is a Demaq-specific implementation of the `fn:trace` function described by the XQuery standard. It's signature is `qs:trace($value as item()*, $label as xs:string) as item()*`. This function can be used to print the textual representation of a sequence of items (value) to the console. The label-string is used as an additional annotation for the serialized items. The function returns the value items without any modification.

```
create rule exampleRule for exampleQueue
enqueue message <example>{qs:trace(//item/description, "Contained descriptions")}</example>
        into output;
```

qs:traceMessage The `qs:traceMessage` can be used to simply write a string to the system console. It's signature is `qs:traceMessage($message as xs:string)`. This function does not return any data.

```
create rule exampleRule for exampleQueue
(
  qs:traceMessage("Entering exampleRule") ,
  enqueue message . into output
);
```

> Note that the rule body in the above example is a sequence of two expressions: the traceMessage function and the enqueue message expression. Thus, in order to incorporate debug messages into the rule, its body has to be changed from a single enqueue expression to a sequence using the brackets.

qs:traceXDM While the `qs:trace` and `qs:traceMessage` functions allow to write textual debug output to the console, they are not really helpful for e.g. printing elements. As explained above, all trace output is simply converted to text (and not properly serialized). For an element, instead of logging a textual representation full element (e.g. `<foo>bar</foo>`), only the textual content (`bar`) will be traced. In order to avoid this, the `qs:traceXDM` function can be used. It takes a sequence of nodes as its input, serializes these nodes, and prints them to the console.

```
create rule persons for input
(qs:traceXDM(//PERSONA),
 qs:traceXDM(.),
 enqueue message <persons>{//PERSONA}</persons> into output);
```

4.8.2 Detecting runtime errors

To facilitate detecting runtime errors (e.g. invalid responses arriving from external systems) it is often helpful to use a gateway queue interacting with the client application as the default error queue (at least during debugging). Thus,

unexpected processing errors and the corresponding error messages do not get lost in the system queue, but can be directly seen. However, writing application-specific error handling code is a far better alternative and this should be done before actually deploying it.

Chapter 5

Application Deployment and Runtime

Once an application is implemented in QDL and QRL, it can be deployed on the Demaq runtime system. This section discusses how deployment works, including a description of the various parameters that can be configured when registering a new application. Additionally, it gives an overview of the various components of the runtime system and their influence on the runtime behavior of an application. Particularly, this includes the garbage collector which removes obsolete messages in regular intervals.

5.1 Deployment Steps

Within this section, we discuss the various steps in the life cycle of an Demaq application. The steps include the creation of a new instance (Section 5.1.1), importing applications (Section 5.1.2), system startup (Section 5.1.3) and shutdown (Section 5.1.4), as well as closing (Section 5.1.5) and, finally, destruction (Section 5.1.6).

5.1.1 Instance Creation

Instance creation is the first step to be performed when deploying an application. When creating a new instance, several instance-specific parameters may be chosen. These parameters remain fixed for the entire lifetime of the instance and can not be altered afterward.

An instance of he `DemaqConf` class is used to provide these parameters to the runtime system. Apart from the name of the application instance, it includes default values for all parameters, making their specification optional. The following parameters may be defined and accessed using corresponding getter- and setter-methods:

InstanceName Each Demaq application instance has a unique name. This name has to be specified upon creation.

The instance name has no default value and must be specified

Logger The runtime system incorporates a logging component that is used to write debug, information and error messages to a logfile or stream. Logging is performed on class level of the C++ implementation. For each class, an individual log level can be chosen, indicating which information should be written to the log. Loglevels 0 and 1 log everything, level 2 logs information and error messages (but no debug information), while loglevel 3 only logs error messages.

In the example below, the loglevels for several classes are set using the corresponding functionality of the Demaq configuration parameter class (in the C++ driver code). For each class, the triple describes the C++ class name the logger is defined in, the target log level (see above), and the target log file to write messages to. In the example below, a single log file (wsx.log) is shared by all class-specific loggers.

```
demaqConf.setLogger(" wsx::Demaq#1#wsx.log"
          " wsx::ThreadManager#0#wsx.log"
          " wsx::RuleExecutor#2#wsx.log"
          " wsx::ActionInterpreter#0#wsx.log"
          " wsx::ActionList#0#wsx.log"
          " wsx::Dispatcher#2#wsx.log"
          " wsx::Dispatcher::ProcessingThread#2#wsx.log"
          " wsx::QueueScheduler#2#wsx.log"
          " wsx::EchoThread#2#wsx.log"
          " wsx::Cond#2#wsx.log"
          " wsx::Mutex#2#wsx.log"
          " wsx::Thread#2#wsx.log"
          " wsx::NatixGateway#0#wsx.log"
);
```

The default value is defined in `demaq.cc`.

GarbageCollectionInterval The runtime system includes a garbage collector that removes unnecessary messages in regular intervals (see Section 5.2.1 for additional details). Depending on the structure and complexity of an application, garbage collection may be a rather expensive operation.

The GarbageCollectionInterval defines when garbage collection should be performed. It is run every k seconds, where k is the GarbageCollectionInterval. Setting it to 0 results in no garbage collection being performed at all.

The default value is 300 seconds (5 minutes).

ConcurrentConnections Any application interacts with remote systems using gateway queues. The runtime system provides corresponding communication facilities for each gateway queue. This parameter defines how many concurrent connection may be handled by each transport protocol endpoint (and thus

by each gateway queue). Resource allocation is performed on demand, thus this value defines an upper bound and has to penalties compared to a lower value in periods of low load (no concurrent connections). This value has to be greater than 0. The smallest valid value is 1, allowing one connection for each gateway queue at a time (particularly, for incoming synchronous protocols this means that no additional connection can be accepted until the reply to the previous connection has been sent).

The default value is 64 connections for each transport protocol endpoint.

TraceStream This parameter defines a C++ stream that should be used to trace system operations (see Section 5.2.2 for details on the tracing facilities of the runtime system).

The default value is 0, indicating that no trace file should be written.

PartitionLogicalName The logical partition name to be used by the underlying message store.

The default value is testpart.

PartitionType The type of the database partition to be used by the underlying message store.

The default value is file, indicating that a file on hard disc should be used.

PartitionPhysicalName The physical partition to be used by the underlying message store.

The default value is test.part.

PageSize The physical database page size to be used by the underlying message store.

The default value is 8192 bytes (8 KB).

LogPartitionSize The size of the recovery log to be used by the underlying message store.

The default value is 3000 pages.

PartitionSize The partition size to be used by the underlying message store for regular data storage (message).

The default value is 1500 pages.

MainMemoryBufferSize The main memory buffer to be used by the underlying message store (buffer pool capacity).

The default value is 1000 pages.

PreserveExisting The PreserveExisting flag indicates whether partitions of the underlying message store should be preserved, avoiding them to be overwritten with newer versions.

The default value is false, indicating that existing partitions should be overwritten.

5.1.2 Importing the Application

After an instances has been created, the application logic can be imported into this instance. Application import causes the required infrastructure for queues, rules, slicings and properties to be created and registered.

5.1.3 Instance Startup

Once application import has been performed, an instance may be started. Starting an instance causes the runtime system to execute the application logic for this particular instance, thus making it interact with remote systems and perform rule execution.

5.1.4 Instance Shutdown

An instance that has been started can be shut down, thus making it stop rule processing and interacting with remote systems.

5.1.5 Closing an Instance

Any non-active instance (either shut down or not yet started) may be closed, thus de-registering it from the runtime system and freeing resources. A closed instance may be opened at a later time and (re-) started.

5.1.6 Destroying an Instance

Once an application is no longer required, the corresponding instance can be destroyed. The destruction of an instance causes the *physical deletion* of all related messages and metadata.

WARNING: Instance destruction leads to permanent deletion of data and cannot be undone.

5.2 Application Runtime

This section discusses specific aspects of the Demaq runtime system, documenting its features and facilities such as the trace log and the interactive debugger. Additionally, the purpose of this section is to provide insight about system internals that may affect the runtime behavior and performance of a deployed application.

5.2.1 Garbage Collector

Demaq includes a two-level garbage collector that removes unnecessary message that are no longer required by application rules. Garbage collection is performed in regular intervals (every k seconds as specified by the GarbageCollectionInterval deployment parameter discussed above). The garbage collector removes

- all processed messages that are no longer referenced by at least one slicing (indicated by the slicings require expression discussed in Section 3.3.4)
- all unprocessed messages that are no longer referenced by at least one slicing and that are stored in queues that do not have any rules (or slicings with rules) defined on them.

Depending on when garbage collection is being performed, the result sequence returned by the `qs:queue` function call may significantly differ.

Garbage collection is performed in two seperate phases. *Slicing garbage collection* removes all unnecessary messages from the persistent representation of slicings (e.g. an index). Thus, a call to the `qs:slice` function does not need to perform message filtering tasks (to filter out messages outside the require window) and may run significantly faster after garbage collection.

After cleaning up all slicings, the second phase of garbage collection is *queue garbage collection*. In this phase, all messages no longer referenced by at least one slice are physically removed from the message store. This allows the system to reclaim storage capacity, as well as reducing the result set of calls to the `qs:queue` function.

Depending on the structure and complexity of an application, garbage collection may be a very expensive operation. This is due to the fact that all slicings and all queues of the application have to be analyzed in order to find candidate messages that can be removed. On the other hand, as discussed above, garbage collection may allow both to reclaim storage capacity and speed up rule execution performance by avoiding or at least simplifying message filtering.

> While garbage collection is being performed, the overall performance of the runtime system may deteriorate. If your application suffers from performance hiccups, you can check the system log to verify whether this may be due to ongoing garbage collection.

5.2.2 System Trace

Optionally, the runtime system writes a trace log reflecting all (high-level) operations being performed (see the TraceStream configuration parameter documentation in Section 5.1.1 for how to activate tracing). This trace includes sending and receiving message, rule execution, application errors and others and can e.g. used for system debugging. The trace stream contains XML elements for each event, the corresponding schema can be found in docs/trace.xsd.

5.2.3 Interactive Debugger

To facilitate application development and error tracking, Demaq includes an interactive debugger (see [8] for details). The Debugger can be used to analyze any Demaq application without requiring any modifications or adaptations to the application code.

Among others, the Debugger includes features for retrieving the components (queues, properties, slicings and rules) of an active application, inspecting the content of the message store (messages, queues, slices, properties) as well as active debugging support using breakpoints, watchpoints and step debugging.

Currently, the only front-end to the Demaq debugger is a command line utility (`ddb.sh`), a visual IDE is under development. The command line utility can be found in the tools folder of the demaq source directory (`src/tools/debugger`) and can be invoked by issuing `sh ddb.sh`. A list of all available commands can then be retrieved from the integrated help menu (`h`).

> The debugger is only available when the corresponding compile-time option is activated. See Section 6.4 for details on activating the debugger and other compile-time options.

Chapter 6

System Installation

In this section, we discuss how to set up the Demaq system. This includes the required software platforms and tools, retrieving the source code, choosing from the various system configurations that are available, and, finally, compilation of the Demaq system.

6.1 Required Third-Party Software Packages

Currently, Demaq compiles and runs of Linux and Mac OS X. It is tested with the latest release of opensuse Linux and Mac OS X. While other operating systems are not supported, it should be possible to compile and run Demaq (and the underlying Natix store) on BSD-based systems with very little effort. A windows port is currently not available due to porting problems with the underlying Natix store.

Demaq requires the following third-party software packages to be installed on the target system.

- libcurl, a HTTP library
- flex, a generator for lexical analyzers
- bison, a parser generator
- Apache libxerces, an XML parsing library
- Natix, a native XML message store
- boost, a package of C++ libraries
- Xalan, an XSLT processor
- Xerces, an XML parser
- Saxon, an XQuery processor

- autoconf, the GNU configuration tool
- automake, the GNU make tool

Optionally, depending on the chosen configuration and features (see below), the following software packages may additionally be required.

- IBM DB/2, a database management system
- qt4, a toolkit for graphical user interfaces
- doxygen, a documentation generation toolkit

6.2 Retrieving the Source Code

The current version of the Demaq source code can be retrieved from the svn repository of the University of Mannheim's database group using the following checkout command:
`svn co svn+ssh://svn@pi3.informatik.uni-mannheim.de/alex/demaq`
As a result, a new directory called demaq will be created in the current working directory (e.g. `/home/someone/`).

6.3 Configuration and Compilation

After checking out the source code, the following steps are required to configure and build the Demaq system. The following examples all assume the location Demaq source code to be `/home/someone/demaq`.

6.3.1 Setting up a build directory

Invoke the automake.sh script in the source directory to set up the build system. While is possible to compile/build Demaq in the source directory this is strongly discouraged. Instead, a seperate build directory should be created, e.g. on a local (non-NFS) disc to speed up compilation, e.g. `mkdir /home/someone/demaq_build`. In the following sections, we assume that this is the location of the build directory.

6.3.2 Configuring the build directory

The next step is to invoke the configure-script in the build-directory. This can be done by issuing `/home/someone/demaq/configure CXX="g++" CXXFLAGS="-g -O0 -W -Wall -Wpointer-arith -Woverloaded-virtual -Winline" CPPFLAGS="-I/home/someone/natix_build/include -I/home/someone/natix/include" LDFLAGS="-L/home/someone/natix_build/lib"` in the build directory.

- `CXX` defines the C++ compiler to be used. This can e.g. be the GNU C++ compiler `g++`, the Intel C++ compiler `icpc` or the icecream distributed compiler wrapper `icecc`.

- `CXXFLAGS` are the command line options that should be passed to the compiler. For example, `-O0` instructs it to perform no optimizations, `-O2` should be used for an optimized build. The above example works fine for `g++`, for `icpc` e.g. `"-w1 -Wcheck -Wdeprecated -Wreturn-type -Wshadow -Wunused-function -Wuninitial ized"` can be used.
- `CPPFLAGS` contains additional flags that should be passed to the compiler. `-Ipathname` instructs the compiler to look for headers in the specified directory. Currently, the Natix build AND source directories (or complete a Natix release) must be included this way.
- `LDFLAGS` contains flags that should be passed to the linker. `-Lpathname` tells the linker to consider a particular path when looking for a library. Currently, the path to libnatix (e.g. in the lib sub-directory of a Natix build directory) should be set this way.
- Optional parameters. Demaq includes several compile time configuration options that can be additionally defined. See Section 6.4 for a detailed discussion of the available options and their effects. Additional documentation can also be retrieved by running `configure --help`

6.3.3 Performing an initial build

After configuration is done, the entire system can be build by running `make`. This will create a fully functional version of demaq. The system tests (located in the `src/tests/system` directory) can be built separately by running make in this directory. These are not included in the default target to speed up compilation.

6.3.4 Setting environment variables

Currently, Demaq relies on Saxon and Saxon extensions for XQuery/DQL processing. For example, the qs:queue and qs:property functions are implemented this way. In order to find the Saxon XQuery processor and the extension functions, the `CLASSPATH` environment variable must include the path to Saxon and the `src/xquery` directory of Demaq. The CLASSPATH can e.g. be extended using `export CLASSPATH=$CLASSPATH:/home/someone/demaq_build/src/xquery:/saxonpath/saxon9.jar`.

Additionally, the `DEMAQ_SRCDIR` environment variable has to be set. Upon system startup, Demaq accesses some library files (most notably the XQuery application library and the debugger infrastructure). In order to properly access these files which are located in the Demaq source-directory, the system recovers the path to the Demaq source from the environment. For this purpose, the DEMAQ_SRCDIR environment variable must be set to the location of the Demaq source (e.g. `/home/someone/demaq/`). This can be done by issuing `export DEMAQ_SRCDIR=/home/someone/demaq/`.

6.4 Compile-time Configuration Options

Demaq includes several compile-time configuration options that affect the features, performance and behavior of the entire system. These configuration options can be enabled or disabled by defining these options when invoking the `configure` command (see Section 6.3.2).

The following configuration options are available.

`--enable-tracing (default: yes)` When this option is set, the system supports to track low-level operations (sending and receiving files, rule execution, errors) in a corresponding trace. The corresponding trace stream these events should be written to can be set as an instance parameter (see Section 5.1.1).

`--enable-auditing (default: yes)` This option defines whether the audit framework of the runtime system should be enabled. This framework allows to track key performance indicators and system runtime information (e.g. number of transactions, etc).

`--enable-logging (default: yes)` If this option is enabled, the Demaq logging framework is available. Using this framework allows to write debugging, information and error messages for individual C++ classes to a corresponding (class-specific) logfile. The individual, class-specific log-level can be set using the runtime Demaq instance configuration mechanism (see Section 5.1.1 for details).

The logging option is available by default for debugging purposes, but should be disabled in performance-critical setups and environments.

`--enable-debugcode (default: yes)` If this option is enabled, the system performs additional sanity checks on internal data structures, return values, control flow and more. This option is very useful for debugging, but should be turned off when doing performance measurements as the runtime performance can be significantly lower with debugging turned on.

`--enable-compilerdebugcode (default: no)` Enabling this option instructs the DQL compiler to perform excessive logging, documenting every rewrite applied and giving additional details abort internal processing steps. As this option produces tons of debug output, it should only be activated if bugs in the DQL compiler have to be tracked.

`--enable-concurrency (default: no)` This option governs whether multiple message store transactions may be active at the same time. If this option is set, the runtime performance of the system may be significantly improved. However, **activating this option significantly changes the runtime behavior of an application** as several messages may be evaluated in parallel. However,

as this is allowed by the Demaq model, this option can be safely activated if the underlying message store allows for concurrent transactions.

`--enable-debugger (default: yes)` In order to support application debugging, the Demaq runtime system optionally includes an interactive debugger.

Section 5.2.3 gives a detailed overview of the Debugger. It discusses the functionality provided, the features supported as well as debugger invocation.

> Enabling the debugger may have a considerable impact on the runtime performance of the Demaq system. It should thus not be enabled in performance-critical environments or when evaluating the system performance.

6.4.1 Building Demaq with IBM DB/2 as Message Store

Instead of Natix, which is the main message store of the Demaq runtime system, IBM DB/2 can alternatively be used for message storage and query execution (see [6] for details).

When the configure command is invoked with the option option `"DB2=yes"`, Demaq is configured to use IBM DB/2. In order to properly use DB/2, a number of additional steps is required.

Installing DB/2 DB/2 (e.g. available from http://www.ibm.com/software/data/db2/express/download.html) must be installed. The Express-C version is sufficient. The database has to be installed with administrative privileges (e.g. root account). Additionally, the db2 binary directory (/bin) must be included in PATH environment variable.

Installing the DB2 Facade The Demaq-specific DB2 glue code is available from it's own source repository. It can be retrieved by issuing `svn co svn+ssh://svn@pi3.informatik.uni-mannheim.de/demaqdb2addon db2transaction`.

Next, the paths in the DB2-Facade have to be adapted fit the local source code and DB/2 paths. These are the first 3 lines in db2transaction/compile.sh.

Additionally, the settings in db2transaction/include/settings.hh potentially have to be adapted.

Finally, the facade can be built using the compile shell script `sh compile.sh`.

In order to allow other components to properly use the DB2 Facade library, the library folders both of DB2 and the DB2 facade have to be included in the LD_LIBRARY_PATH. These are `lib32` and `db2transaction/lib/.libs`, respectively.

Installing Natix with DB/2 Extensions The DB/2 message store adapter inherits some features and code from Natix. Thus, in order to support DB/2, the Natix message store has to be installed and has to be compiled with a special DB/2 option in order to provide the necessary features. This can be done by

adding the `db2` option to the Natix configure command (e.g. `abusy --config natix gcc debug demaq db2`).

Potentially, some Natix Makefiles have to be adapted in order to include the proper paths (these are `(src/schema/physical/db2document/Imakefile)` and `src/schema/physical/query/saxon/Imakefile`).

Configuring Demaq Apart from activating the DB/2 option using `"DB2=yes"`, the paths to the DB/2 include directory, the DB2 Facade, the DB/2 libraries and facade libraries have to be included when configuring the Demaq system. This can e.g. be done using `/home/someone/demaq/configure DB2=yes CXX="g++" CXXFLAGS="-g -O0 -W -Wall -Wpointer-arith -Woverloaded-virtual -Winline" CPPFLAGS="-I/home/someone/natix/include -I/home/someone/natix/include -I/opt/ibm/db2/V9.5/include -I/home/someone/db2transaction/include" LDFLAGS="-L/home/someone/natix_build/lib -L/opt/ibm/db2/V9.5/lib32 -L/home/someone/db2transaction/lib/.libs"`

> The message store implementation based on IBM DB/2 is currently not complete. It lacks many important features such as support for gateway queues, slice require expressions, message garbage collection and more. When using DB/2 as message store, the system will most likely be unable to provide essential functionality for message processing. Using DB/2 is thus currently *strongly discouraged*, except for experiments.

6.5 Speeding up the build process

The build system uses libtool to simplify building the Demaq library on multiple platforms and to provide dynamic and static library versions. By default, both static and dynamic libraries are built, thus effectively doubling the effort required by the build system.

In order to avoid this overhead, the build system can be configured to only create either a static or dynamic version of the Demaq library, the individual sub-libraries and object files. This can be done by using the `--disable-shared` or `--disable-static` options when invoking the configure script.

Chapter 7

System Architecture

This chapter gives a brief overview of the components of the Demaq system. This includes the Demaq query compiler (Section 7.1), which transforms application specifications (written in QDL and QML) into execution plans for the runtime system (Section 7.2).

7.1 Query Compiler

Before an application can actually be run by the Demaq server, the query compiler is used to transform the textual application representation into an execution plan. The query compiler performs normalization and optimizes the rule set. Thus, the final execution plan might differ substantially from the initial application specification. For example, rules operating on the same queue may be combined, intermediate queues may be removed, or expressions in the rule body may be replaced by other, logical equivalent ones.

The query compiler operates in several phases, starting with the syntactical analysis of the input rules using a lexer and parser. If no errors are found in the input application, this first phase results in an Abstract Syntax Tree (AST) reflecting the application specification. Afterwards, additional checks are performed in order to verify the semantical correctness of the AST with respect to the QDL and QML language definitions. Subsequently, normalization and optimizations are performed by rewriting the AST into another, logically equivalent one. Finally, this optimized AST is transformed into the execution plan for the Demaq runtime system.

7.2 Runtime System

The runtime system consists of three main components: An XML message store for queue management and persistent data storage (Section 7.2.1), the runtime core (Section 7.2.2) which performs message processing and rule execution, and

the communication system (Section 7.2.3) providing communication channels to external systems. Figure 7.1 gives an overview of this three main components.

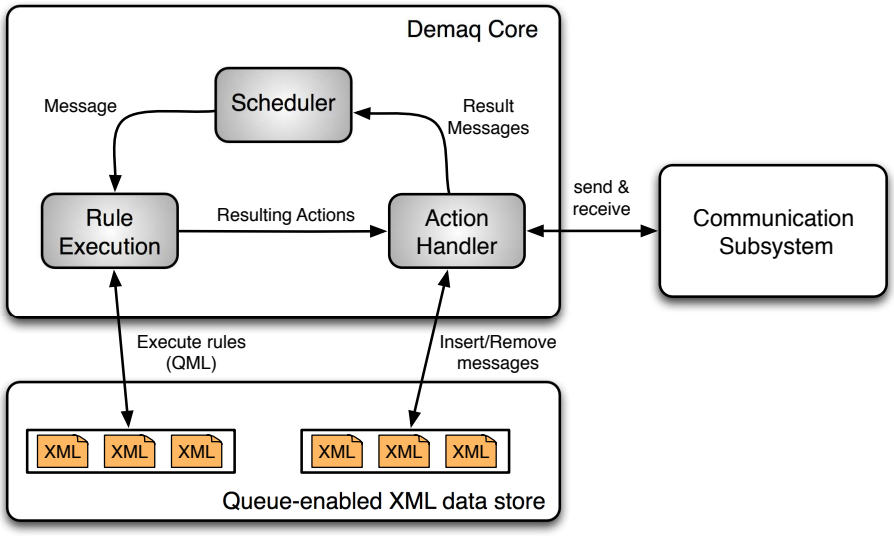

Figure 7.1: The Demaq runtime system

7.2.1 XML Message Storage

An native XML database management system provides the foundation of the runtime system. It is responsible of the efficient and reliable management of XML messages, as well as for efficient rule execution on (sequences of) messages.

To achieve high performance, the idea of the current Demaq design is to perform rule processing inside the database kernel, thus reusing the query processing facilities of the database system, and avoiding data transfer into the runtime core.

7.2.2 Runtime Core

Operating on top of the XML message store, the runtime core governs all active functionality of the Demaq system. It incorporates the scheduler that decides which message has to be processed next. For this message, a rule execution component computes the result produced by application rules (e.g. new messages that have to be enqueued into other queues, or messages that have to be sent), e.g. by invoking the query execution component of the message store. The rule execution component produces a list of actions that have to be performed, and which are executed by a corresponding action handler component. This component then e.g. enqueues messages, or causes a message to be sent to the communication system for sending it to a remote transport endpoint. Finally,

the consequences of action handling (e.g. new local messages that have to be processed subsequently) are announced to the scheduler component.

7.2.3 Communication System

All communication with external transport endpoints is performed by the communication system. It implements the corresponding low-level protocols (HTTP, SMTP, POP3). Every remote transport endpoint is represented by a *communication channel*, which allows the runtime core to transfer messages to an associated external system. Every individual connection to an external system is represented by an individual *call*. Calls are (depending on the transport protocol) bidirectional connections that can be used for data transfer.

7.3 Visual Editor

To simplify application development, Demaq includes a visual editor. The editor can be used to quickly implement an application using a simple drag-and-drop interface and a syntax aware rule editor. Additionally, it includes visualization features to analyze Demaq trace files as well as a visual interface to the Demaq debugger.

The editor can be found in a separate code repository (`demaqeditor`) and has its own build system, much similar to that used by the Demaq system. Please see the editor project documentation for additional details.

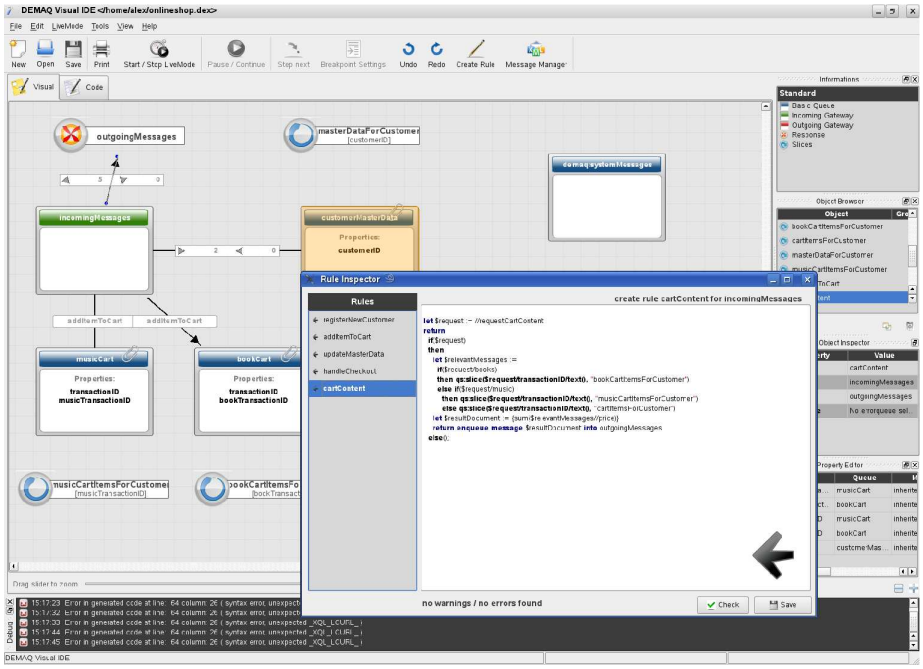

Figure 7.2: Visual Editor

Chapter 8

Implementation

This section discusses some of the implementation details of Demaq and the corresponding extensions that have been made to the underlying Natix system. This section is intentionally very brief as the source code provides extensive documentation about the implementation details.

8.1 Query Compiler

The query compiler relies on a combination of lexical analyzer (flex-generated) and a parser (bison-generated) to construct the AST. Once complete, this AST is traverse multiple times to perform semantic analysis and normalizing/optimizing AST rewrites.

8.1.1 Query Rewrites

In the current implementation, most important are the normalizations, which transform the rule bodies of Demaq rules into XQuery 1.0, particularly without any update statements or Demaq extensions (apart from system-provided functions such as qs:queue). Additionally, the corresponding rewrite module of the compiler (traversal) performs some optimizations and extensions to simplify and speed up processing for the runtime system. Each individual normalization and optimization can be selectively turned of, depending on the features provided by the runtime system or other requirements. The following sections give a brief overview of the individual rewrites that are supported by traversal.

Normalize qs:queue function parameters The `qs:queue` function expects the name of the target queue as the only parameter. In rules defined on queues, this parameter may be omitted in application rules. This rewrite normalizes all qs:queue function calls, making sure that each of them has the name of the target queue as the parameter. This is done by adding the name of the queue the rule is defined on the function call (for rules defined on queues) or raising a

semantic error (for rules defined on slicings). The corresponding rewrite mode is `NORMALIZE_QUEUE_FUNCTIONS`.

Normalize qs:slice function parameters The `qs:slice` function expects two parameters: the slicekey as the first and the name of the target slice as second parameter. Syntactic shortcuts allow either omitting the name of the slice (in rules defined on slicings) or even both parameters in application rules. This rewrites expands this syntactical shortcuts, making sure that each qs:slice function call has exactly two parameters afterward. This is done by inferring the slicekey by calling the `qs:slicekey` function and the slicing name from the rule target (this only works for rules defined on slicings, if the rule is defined on a queue this is a semantic error).

The corresponding rewrite mode is `NORMALIZE_SLICE_FUNCTIONS`.

Normalize qs:slicekey function parameters Similar to parameter expansion for the `qs:slice` function discussed above, this rewrite ensures that each qs:slicekey function call has exactly one parameter (the name of the target slicing). For rules defined on slicings, this parameter can be omitted, defaulting to the target of the rule. For such parameterless qs:slicekey functions, this rewrite infers the slicing name from the rule target (calling the qs:slicekey function without parameter in a rule defined on a queue is a semantic error).

The corresponding rewrite mode is `NORMALIZE_SLICEKEY_FUNCTIONS`.

Remove qs:slice function calls from rule bodies This rewrite can be used to remove all qs:slice function calls from rule bodies, replacing them with semantically equivalent expressions using the qs:queue and qs:property functions. This rewrite is particularly useful for runtime systems that do not or only partially support slicings and related access operations.

The corresponding rewrite mode is `REMOVE_SLICE_FUNCTIONCALL`.

Remove rules defined on slicings This rewrite removes all rules defined on slicings and introduces additional rules for all queues the slicings are defined on. Thus, while preserving the semantics of the application, the target runtime system is not required to support rules on slicings.

The corresponding rewrite mode is `REMOVE_SLICE_RULES`.

Merge all rules defined on a conjoint queue In this rewrite, all rules defined on the same queue are combined into a single, large rule. In order for the rewrite to be applied, the rules also have to share the same error queue.

The corresponding rewrite mode is `MERGE_QUEUE_RULES`.

Remove qs:slicekey function calls from rule bodies This rewrite can be used to replace all calls to the qs:slicekey function with semantically equivalent to the qs:message and qs:property function. Note that this rewrite expects all qs:slicekey functions to have exactly one parameter, thus slicekey function

normalization should be activated, too. This rewrite is particularly useful for runtime systems that provide no support for the slicekey function.

The corresponding rewrite mode is `REMOVE_SLICEKEY_FUNCTIONCALL`.

Normalize qs:property function parameters The `qs:property` function may be called with two parameters, the message to retrieve the property for and the name of the requested property. The message the property should retrieved for may be omitted, defaulting to the context message (`qs:message`) of rule execution.

This rewrite expands all single-parameter qs:property function calls by adding a call to the qs:message function.

The corresponding rewrite mode is `NORMALIZE_PROPERTY_FUNCTIONS`.

Optimize message copying / forwarding using message links Sometimes, application rules may just copy one message from one queue to another, e.g. depending on the type of message received. As copying may be an expensive operation, a runtime system may choose to just enqueue a reference to the existing message (already stored somewhere in the message queues) instead of really copying it (much similar to a link in a file system). In some cases (such as `enqueue message qs:message() into someQueue`, the query compiler may detect message copying. In these situations, this rewrite replaces the enqueue operation with another operation called `link`, allowing the runtime system to avoid the expensive copy operation.

The corresponding rewrite mode is `OPTIMIZE_ENQUEUE_LINKING`.

Piggyback slicekey computation when enqueuing messages In Demaq, message access is mainly based on the concepts of slices. Thus, a runtime system has to provide mechanism to access all messages in the same slicing, sharing the same slice key. As slicekey computation involves evaluating an XQuery expression on a particular message, this typically is a rather expensive operation.

This rewrite can be used to rewrite application rules in order to automatically determine all relevant slicekeys when enqueuing a message. The property expressions for all slicings defined on the queue the message is inserted are evaluated and added as additional (system) properties. These properties can e.g. be materialized in the message store or also used by the runtime system to build indexes supporting slice-based message access.

The corresponding rewrite mode is `COMPUTE_SLICEKEY_VALUES`.

Remove all slicing definitions from application This rewrite simply removes all slicing definition expressions (`create slicing...`) from the application. This rewrite is particularly useful in conjunction with the other slice-removal rewrites to allow deploying a Demaq application on a runtime system with no slicing support.

The corresponding rewrite mode is `REMOVE_SLICING_DEFINITION`.

Generate additional receive rules for incoming gateway queues This rewrite generates additional rules for receiving messages from communication channels into gateway queues. Using this approach, all incoming messages are evaluated by the rule execution system, e.g. allowing to compute slicekeys for messages before inserting them into gateway queues (see above). Additionally, no rules have to be hard-coded into the runtime system.

The corresponding rewrite mode is `GENERATE_GATEWAYQUEUE_RECEIVE_RULES`.

Generate additional send rules for outgoing gateways This rewrite is much similar to the one for generating receive rules (described above). The difference is that this rewrite creates rules for outgoing gateway queues, instructing the runtime system to perform messaging actions whenever a message is enqueued.

The corresponding rewrite mode is `GENERATE_GATEWAYQUEUE_OUTGOING_RULES`.

Generate XQuery filter functions for slice require expressions As discussed in Section 3.3.4, slicing definition includes a `require` expression that defines which parts of a slice have to be retained. Using this rewrite corresponding filtering functions are defined and added to each call of the qs:slice function. Thus, rule evaluation automatically filters out those messages that should not be seen for this particular slice, but are still kept in the message store.

This rewrite is particularly useful for runtime systems that do not support message deletion or provide lazy, deferred maintenance algorithms (e.g. offline garbage collection).

The corresponding rewrite mode is `GENERATE_SLICE_REQUIRE_FILTERS`.

Generate property definitions for system properties Demaq provides several system-defined properties that can be used by application rules. Most important (and the only system properties currently provided) are transport protocol parameters that are included for message received using incoming gateway queues. In order for the runtime system to properly handle these parameters, this rewrite generates property definition statement for all system properties and all incoming gateway queues. The properties are defined as `inherited` to allow propagation from the communication channels to the corresponding queues.

The corresponding rewrite mode is `GENERATE_SYSTEMPROPERTY_DEFINITIONS`.

Supporting slice access function calls to variable target slicings By default, the compiler enforces all calls to the qs:slice function call to include the name of the target slicing as a constant value. This allows for extensive compile time reasoning and semantic application analysis. However, the Demaq language specification allows for the definition of the target slicing as a variable in principle, and this feature may be required by an application. For example, the interactive debugger may not operate correctly without support for variable

slice function call targets. Thus, using this rewrite, dynamic slice function call targets may be enabled.

The corresponding rewrite mode is `ENABLE_VARIABLE_SLICEFCTCALLS`.

Generating Optimized Require Filter Expressions When using filter functions for filtering out messages that belong to the irrelevant prefix of a slice, the filter function needs to be evaluated every time a slice is accessed in order to preserve slicing semantics. The filter function has to potentially evaluate the require expression for all prefixes of all suffixes of the input sequence. Depending on the complexity of the require expression being used, this expensive analysis operation can be replaced with a simpler filter function that yields the same result but can be evaluated more efficiently.

For example, require expressions that implement message windows based on the number of messages to be contained in a slice can be evaluated on complete suffixes of the input sequence instead of checking each prefix for all suffixes. Examples of require expressions that can be handled this way include `count(qs:history()) eq 5` or `count(qs:history()/elementName) gt 3`.

Another optimization opportunity exists for require expressions that always yield a constant value, independent of the current message history. For example, an application may use a require expression of `fn:false()` to indicate that the messages in a particular slice should never be removed from the message history. For constant values indicating that the entire history of a slicing needs to be retained, the filter function may directly return all of its input nodes.

If this rewrite is active, the compiler tries to replace the full-fledged require filter expression with simpler variants wherever possible. This rewrite may dramatically improve the runtime performance of an application.

The corresponding rewrite mode is `OPTIMIZE_REQUIRE_FILTERS`.

Normalizing Slicing Property Definitions The Demaq language includes the `create slicing property` shortcut with allows to create a slicing and a corresponding property definition using a single, combined expression. This rewrite replaces this combined expression with both a separate slicing and a property definition. Thus, while preserving application semantics, this normalization may simplify subsequent code analysis and rewriting steps.

The corresponding rewrite mode is `NORMALIZE_SLICINGPROPERTY_DEFINITIONS`.

8.1.2 AST Serialization / Execution Plan

Afterwards, the AST is serialized into an XML representation (called DQLX). This representation contains XML representation for all QDL constructs. The XQuery fragments of the rule bodies are kept in their textual form and are represented as CDATA fields. For convenient access of DQLX representations, Demaq incorporates a SAX-based parser which can be used to transform DQLX documents into main-memory objects.

8.2 Runtime System

8.2.1 XML Message Storage

Most of the XML handling, transformation and query execution code is provided by the Natix system, which is used as the XML data store (XDS) of the current Demaq implementation. The Demaq runtime system relies on the Natix C++ API to invoke the corresponding functions. Most of the Natix-specific access operations are implemented by the corresponding `NatixGateway` class, with the exception of the rule execution component and the corresponding `ActionList` class, containing the results produced by the rule execution component.

There are several, Demaq-specific extensions in the Natix XDS. First and foremost, these are native queuing operations that have been added to the Natix kernel in order to allow performing `enqueue` and `dequeue` operations. Additionally, there are some special purpose views (such as the `ChildElementSequenceView`, which allows to iterate over all child elements in a particular `DOMView` as if they were seperate DOM documents) that allow for more convenient and high performance processing in Demaq.

Unfortunately, the Natix XDS does currently neither incorporate an XQuery compiler nor execution system. Thus, the most significant extension has been the integration of the Saxon XQuery engine into Natix that is used for evaluating XQuery fragments. To provide high-performance processing, Natix uses a Java-front-end to Saxon and relies on named pipes for data transfer from and to this front-end. The Demaq-specific access operations (such as qs:queue) are implemented in Java as user-defined functions, that use a simple, textual and callback-based protocol to access data from Natix storage. This way, arbitrary messages and properties can be dynamically accessed by Saxon.

8.2.2 Runtime Core

The runtime core realizes all the active processing that is done by the Demaq server. It interacts with the two other main components, the message store and the communication system. Upon startup, the runtime core initializes the underlying message store and main-memory structures according to the application specification in the DQLX file created by the query compiler.

Whenever a message arrives from an external source at the communication system, the core uses `CommunicationChannels` to exchange data with the communication system. Every incoming connection is reflected by a corresponding `Call`. The runtime core then stores the corresponding message payload (contained in the Call) in the message store.

Every message (either received from an external system or resulting from local rule execution) is announced to the `QueueScheduler`, that keeps a list of messages that still have to be processed by evaluating the corresponding application rules on it. The runtime core hosts several `ProcessingThreads` which perform rule execution (in parallel). These ProcessingThreads are managed by `Dispatchers` which contain groups of related threads (i.e. having the same

processing purpose).

Typically, every processing thread requests the next message to be processed from the QueueScheduler. Afterwards, it invokes the `RuleProcessor` (that can be either a `NatixRuleProcessor` or a `DB2RuleProcessor`) to evaluate the application logic for this particular message. It receives a list of `Actions` to be performed from the RuleProcessor. These actions are then interpreted by the `ActionInterpreter` (that can be either a `NatixActionInterpreter` or a `DB2ActionInterpreter`) which performs the corresponding operations (e.g. enqueues a message, or sends data to the communication system). The interpretation of every message leads to a number of `Consequences` which describe how the action execution affects the runtime core (e.g. the execution of an enqueue action has the consequence that the new message has to be announced to the QueueScheduler). Finally, this consequences are announced to the Dispatcher, which announces them to the corresponding QueueScheduler. These processing steps, starting with receiving a message handle from the QueueScheduler and ending with announcing the consequences are sometimes referred to as a *processing cycle* or *processing loop*.

8.2.3 Communication System

The communication system is implemented by a series of special purpose threads, each of them implementing a particular transport protocol (e.g. HTTP, SMTP, POP3). As these threads provide the Demaq system with access to external systems, they are often referred to as *external threads* in the Demaq documentation.

External threads are managed by a corresponding thread manager and use communication channels and calls to communicate with the threads performing the operations in the runtime core (*internal threads*).

8.3 Test Framework

Demaq incorporates a test system based on the DejaGnu test framework. It supports both C++ unit testing as well as complete application testing. Unit tests are most useful to verify that a C++ component in Demaq is working as expected. The main purpose of application tests is to check whether a given DQL application behaves correctly.

All tests that are part of the test system are located in the `src/tests` directory. In order to be recognized by the test framework, all tests must be located in a folder with a `demaq.` prefix (e.g. `demaq.httptests`).

8.3.1 Running Tests

The test system is integrated into the Demaq build system. Thus, it can be easily invoked by running `make check` in the build directory. While running,

the test system writes progress information to the system console and records detailed trace information for each tests at the corresponding test location.

After all tests have been run, the number of successful and failed tests are written to the system console. Detailed information which tests failed and why is logged in special test summary file (`demaq.sum`) that can be found in the `src/tests/` directory of the build folder. An XML version of the test summary file is also available (`demaq.xml`).

8.3.2 C++ Unit Tests

The C++ unit testing facility allows to easily create unit tests. Unit test can e.g. be used to verify that a particular C++ class of the Demaq system is working as expected. The idea is to create a small C++ testdriver application that interacts with the class or classes to be tested. This testdriver can then communicate the results to the test system. This can be easily done by creating an instance of the `wsx::Test` class, and invoking the `pass`, `fail`, `untested` or `unresolved` methods to indicate the test result. Multiple tests can be handled by a single instance of the `wsx:Test` class.

A very simple example how to create a C++ unit test can be found in the `src/tests/demaq.dummytest` directory. This example can be used easily as a template for developing new unit tests.

Apart from the C++ file, a corresponding expect-file is required in order to run the test as part of the testing framework. The expect file (with a `.exp` suffix) is required to invoke the binary created from the C++ source code of the testdriver file. It should simply use `host_execute` with the binary name as the only parameter.

8.3.3 Application Tests

Application tests can be used for a variety of testing purposes. Most important is to verify that a particular Demaq application behaves as expected. For this kind of test, the steps to perform have to be completely scripted in the expect language. The expect file is then executed by the test system to run the test.

For Demaq, there are several common patterns how tests are build. Usually, application tests start a Demaq server for a given DQL file, and send some messages to the server in a next step. Finally, the server (or client) output is compared against a file containing the expected output in order to check whether the system behaves as expected. To simply create tests following these typical patterns, the test system provides a library that can be used to run such a test with a single function call. A detailed list of the supported functions can be found in the library that is located in the `src/tests/lib` folder.

To create a new application test, a corresponding expect file has to be created. Several examples can be found in the `demaq.basictests` directory. The expect file should first import the Demaq test library (using `load_lib demaq_test.exp`). Afterwards, the test framework can be properly used.

Apart from using the functions provided by the Demaq test library, the complete functionality provided by the expect language can be used to implement the test.

Chapter 9

Legion Application Distribution System

Legion (also known as *TransScale* or *Autoscale*) is a system for the semi-automatic distribution of a Demaq application to a cluster of machines, each of them running the Demaq runtime system. Legion is a source code translator that transforms the initial application specification into a set of application programs that can then be run on a cluster of available machines.

The underlying idea is to move those parts of an application that are independent from each other to different hosts, and convert local message flow in the initial application into remote messaging operations. This allows to distribute the application logic without the need of creating a distributed runtime system.

As this manual aims at providing documentation for the Demaq programming model and languages, as well as the corresponding runtime system, we do not discuss details of the Legion system here. Instead, we refer the interested reader to the corresponding papers [2, 4] for an in-depth discussion.

Legion consists of the following four key components that can be found in the `src/legion` directory of the Demaq distribution.

9.1 Dependency Analysis

Dependency analysis identifies the data dependencies between the queues of an application by analyzing the `qs:queue` and `qs:slice` message history access operations within application rules. These data dependencies restrict the number of individual fragments an application can be decomposed into (see `phase1.cc`).

9.2 Host Allocation

The goal of host allocation is to assign the individual application parts to the available machines in a way that equally distributes the expected application

workload, and at the same time minimizes the communication between these hosts. Host allocation provides several heuristics to approximate and optimal solution within reasonable time. It relies on profiling information that reflects the expected application workload to derive a reasonable distribution (see `allocation.cc`).

9.3 Code Generation

Code generation transforms an initial application specification into a set of standalone sub-applications. These sub-applications can then be deployed on a cluster of machines. The code generation set relies on a valid assignment of application fragments to hosts, produced by a previous allocation step (see `phase3.cc`).

9.4 Scalability Transformations

Typically, the message history access operations (and the data dependencies induced by them) obstruct the distribution of an application to more than a few host machines. To increase the distribution potential of an application (and thereby improve scalability), *scalability transformations* can be used. They transform the initial application specification using source-level rewrites (and also adapt the expected workload profile to reflect application changes) (see `rewrite.cc`). We refer the interested reader to [2] for a discussion of the various rewrites that are provided by the Legion system and of the benefits and drawbacks of their application.

Chapter 10

Further information

This chapter contains some additional references that help to understand the Demaq system and programming language.

- Both QDL and QML are based on the XQuery language standardized by the W3C [1]. The standard contains a detailed explanation of the expressions and constructs of XQuery, including several examples as well as a grammar definition in EBNF. Reading the XQuery standard will dramatically help to understand the Demaq programming language.
- The syntax and semantics of the `enqueue message` expression are based on the XQuery Update Facility [5], as proposed by the W3C. The mechanism how the list of pending messages are constructed and incorporated into the message store is also very similar to the XQuery Update Facility.
- The CIDR 2007 paper [3] gives an overview of the Demaq system from a more abstract perspective and discusses how Demaq relates to other approaches.
- There are several small text files illustrating particular aspects of the system, e.g. containing a EBNF definition of QDL and QML. These files are located in the /doc directory of the Demaq repository.
- Demaq uses doxygen to generate a easy-to-read documentation from the source code. This documentation can be generated by calling "make doc" in the Demaq top level build directory.
- For Natix, there are several publications available at http://db.informatik.uni-mannheim.de/ as well as a detailed overview of the Natix API.

Appendix A

Frequently Asked Questions

This chapter contains a list of some questions that may arise in the Demaq context. It contains both questions regarding the QDL and QML programming languages, as well as some details of the current implementation.

A.1 General

Which platforms can I use for running Demaq on them? Currently, the Demaq system can be run on Linux platforms and on BSD Unixes with Intel CPUs. Demaq is tested regularly on opensuse and Mac OS X.

A.2 Application Developers

Is there any formal definition of the syntax of QDL and QML? A definition of the grammar of both QDL and QML in EBNF is available in file docs/dql_bcnf.txt.

Is there any editor support for creating Demaq applications? Demaq includes a visual editor which can be found in a separate code repository (see Section 7.3 for details). Additionally, there is a language definition which can be used for syntax highlighting in vim, which can be found in file docs/dql.vim.

I receive XQuery errors from the runtime system, however the code presented in the error is different from my application code? The XQuery code used by the runtime system may differ significantly from the application code, as the Demaq query compiler performs optimizing rewrites and normalizations. You can inspect the code generated by the query compiler and used by the runtime system. It is available in the file dqlxoutput.xml, located in the directory the Demaq server has been started from. The visual editor also provides an import feature which allows to visualize and edit the dqlx file created by the query compiler.

Can I make the Demaq system perform operations upon startup, without receiving an external message? You can create a corresponding rule on the Demaq system messages queue (demaq:systemMessages). Whenever an application is started (or terminated), a corresponding message is inserted into this queue.

The execution performs unexpected operations for a Demaq application. Are there any tracing facilities that help to understand the steps executed by the runtime system? The runtime system optionally performs extensive logging. This includes the system log containing debug messages for the various classes and components (usually created as wsx.log). The content of the system log may be customized by setting corresponding log levels in the DemaqConfig upon system startup.

Additionally, a trace stream may be registered at DemaqConfig, which lists all actions that are performed by the system together with their timestamp and context. This trace stream may be graphically visualized by the Demaq editor live mode.

Last not least, Demaq includes an interactive debugger that provides powerful application analysis and control features. See Section 5.2.3 for details.

A.3 Demaq Hackers

Apart from this manual, is there additional documentation for the source code? An extensive source code documentation can be generated using doxygen. You can do this by calling "make doc" in the build directory.

Are there any requirements for adding new code to the repository? You should review the coding style conventions listed in file docs/codingstyle.txt. Additionally, to allow creating a useful doxygen documentation, your code should include the corresponding comments for all classes and methods. You should never commit code to the repository that renders the Demaq system broken or causes the entire project to fail to compile.

Are there any automated/nightly builds and tests? No, unfortunately not. This still has to be implemented.

May I port Demaq to another architecture, e.g. Windows? Yes, of course. Please note that the Demaq runtime system currently heavily depends on the Natix database management system, which is only available for Linux and Mac OS X. Thus, porting Demaq to another platform, e.g. windows thus involves porting Natix, too, which is not trivial, at least for the windows platform.

Bibliography

[1] S. Boag, D. Chamberlin, M.F. Fernandez, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. Technical report, World Wide Web Consortium, January 2007. W3C Recommendation.

[2] Alexander Böhm and Carl-Christian Kanne. Scalability transformations on declarative applications. Technical report, University of Mannheim, 2009. `http://db.informatik.uni-mannheim.de/publications/TR-09-003.pdf`.

[3] Alexander Böhm, Carl-Christian Kanne, and Guido Moerkotte. Demaq: A foundation for declarative XML message processing. In *CIDR*, pages 33–43. www.cirdb.org, 2007.

[4] Alexander Böhm, Erich Marth, and Carl-Christian Kanne. Transscale: Scalability transformations for declarative applications. In *Proceedings of the 26th International Conference on Data Engineering (ICDE), Long Beach, California, USA.*, 2010. Demonstration.

[5] Don Chamberlin, Daniela Florescu, Jim Melton, Jonathan Robie, and Jérôme Siméon. XQuery Update Facility 1.0. Technical report, World Wide Web Consortium, August 2007. W3C Working Draft.

[6] Dennis Knochenwefel. Integration eines relationalen Datenbanksystems als Nachrichtenspeicher in das Demaq-Ausführungssystem. Master's thesis, Universität Mannheim, September 2008.

[7] Andreas Krämer. Design und Implementierung eines Modul-Konzepts für Demaq. Master's thesis, Universität Mannheim, April 2009. In german.

[8] Martin Kremer. Ein Debugger für Demaq. Master's thesis, Universität Mannheim, September 2008.